THE BASIC ADAM®

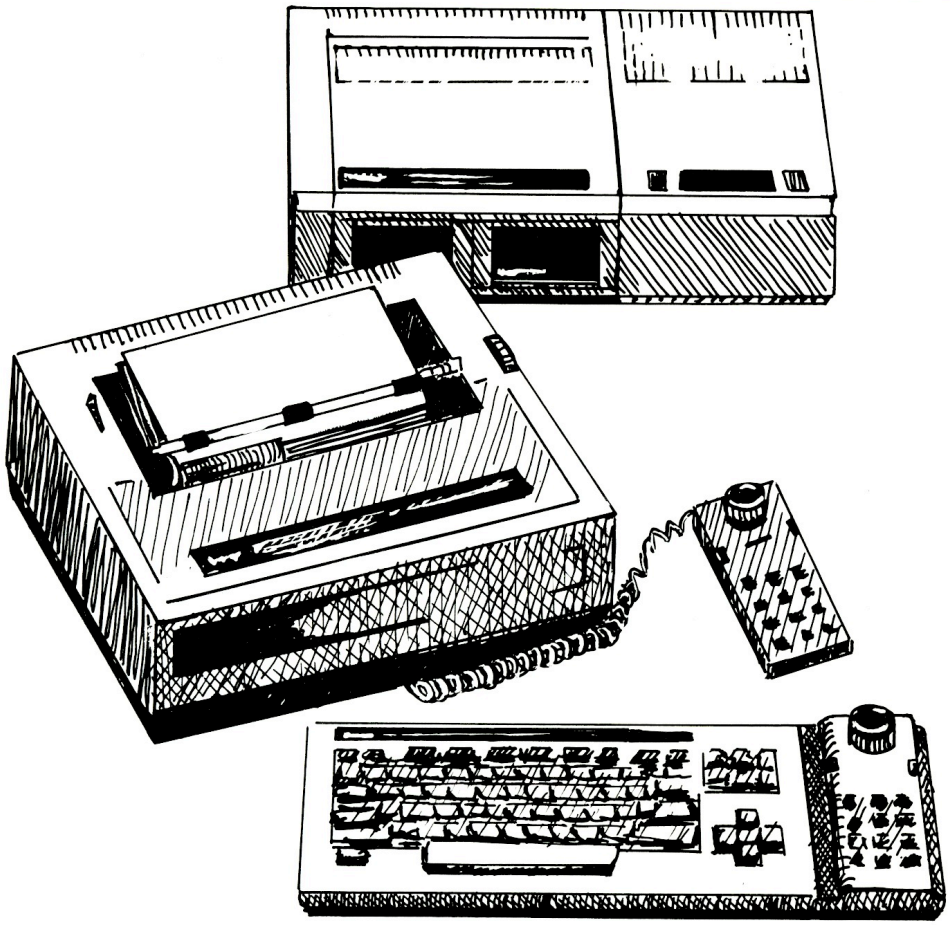
A SELF-TEACHING GUIDE



WILLIAM ABIKOFF
GARY CORNELL

STG

MR ROBERT STROUD
231 STORMONT RD
VICTORIA BC
V9B 1P7



The Basic ADAM

William Abikoff
Gary Cornell

A Wiley Press Book
John Wiley & Sons, Inc.
New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Judy V. Wilson
Managing Editor: David Sobel
Editor: Theron Shreve
Book Production Services:
The Publisher's Network

ADAM™ is a trademark of Coleco Industries, Inc. © 1984 Coleco Industries Inc.

SmartBASIC™ is a trademark of Coleco Industries, Inc. © 1984 Coleco Industries Inc.

Donkey Kong™ is a trademark of Nintendo of America Inc. © 1981, 1983 Nintendo of America Inc.

BUCKROGERS™ and SEGA® are trademarks of SEGA ENTERPRISES INCORPORATED © 1982 . SEGA ENTERPRISES INC.

MONOPOLY® is Parker Brother's registered trademark for its real estate trading game equipment and is used with its permission.

Copyright© 1984 by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by Section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging in Publication Data

Abikoff, William.
The basic Adam.

Includes index.
1. Adam (Computer)—Programming. 2. Basic (Computer program language) I. Cornell, Gary. II. Title.
QA76.8.A153A25 1984 001.64'2 84-7532

ISBN 0-471-80807-5

Printed in the United States of America

84 85 10 9 8 7 6 5 4 3 2 1

Table of Contents

PREFACE		**ix**
Chapter 1	**Getting into ADAM**	**1**
	The Components of ADAM 2	
	Hints and Precautions 15	
	Using your ADAM 19	
	Up and Coming for ADAM 21	
	BUZZwords 24	
Chapter 2	**What is a Computer?**	**25**
	An Office as a Model for a Computer 26	
	The Inside of ADAM 28	
	Computer Languages 33	
	Summary 36	
	BUZZwords 37	
Chapter 3	**ADAM is a Calculator**	**39**
	Simple Calculations 40	
	Exponentiation and E Notation 43	
	Summary 52	
	BUZZwords 53	

Chapter 4 **Statements in BASIC** **55**
Introduction to Programming 55
Our First Program 56
Translating the Solution to a Problem into a Program 61
Assignment Statements 66
More Sophisticated Programs and How to Save Them 70
Some Quick Words About Debugging a Program 75
Summary 76
BUZZword 77

INTERLUDE 1 **79**
Editing Programs 79
BUZZwords 88

Chapter 5 **Changing the Order of a Program** **89**
(Always) Changing the Order 90
You Decide When to Change the Order 96
Changing the Order (Sometimes) 101
IF-THEN 107
Keeping Count: LOOPING 117
Summary 122
BUZZwords 122

Chapter 6 **Doing Things Repeatedly** **125**
Repeated Operations - A Second Look 125
Doing it by Steps 140
FOR-NEXT Loops Inside Each Other 145
Summary 151
BUZZwords 151

Chapter 7 **Pictures** **153**
Getting Started 154
More on Pictures 159
A Bar Graph Program 167
The Low-Res Graphics Editor 171
Summary 178
BUZZwords 179

SmartBASIC's File Handling Commands 441
Summary 445
BUZZwords 446

Afterword - Playing in ADAM's Brain **447**
Using the Program 470

Solutions to Self-Checks **477**

Appendix A - The ASCII Codes **491**

Appendix B - Error Codes and Messages **496**

Appendix C - Mathematical Formulas **499**

Appendix D - The Difference Between SmartBASIC and Applesoft BASIC **501**

Glossary **505**

Reference Guide **511**

Index **521**

Preface

The ADAM Home Computer System's built-in features include an electronic typewriter, a word-processing system and even two different ways to play video games. First and foremost ADAM is a powerful computing system. ADAM also comes with a computer language called SmartBASIC. In order to master ADAM rather than just use some of its features, you have to gain control of ADAM's computer. By programming in SmartBASIC, you can do just that. The programs you can create will tell ADAM what to do.

The purpose of this book is to show you how to program in SmartBASIC. No background is needed to use the book - except a little familiarity with a typewriter keyboard. Patience is needed though, both because nothing worthwhile can be learned quickly and it's easy to get frustrated when a little typo makes your program go wild.

Since algebra or other kind of mathematics is needed to become a good programmer, we don't spend much time on using ADAM to solve math or scientific problems. We concentrate more on how to use ADAM to help in real-life situations and problems and, also, to have a good time by drawing and playing games.

To learn programming, you have to sit down at the keyboard and work through the book. There are lots of questions which will help you check yourself as well as programs that might be fun to write. The questions let you see if you have mastered the material and the programs may give you ideas for your own programs. We don't think it's a good idea to skip around in this book. Many of the techniques that we show you in one chapter are used in later chapters.

Once you have learned to program in one language on one computer, it's

The Basic ADAM

The first problem you face is getting the inner box out of the carton. The top of the inner box, the one with the colorful design, is glued to the bottom of the outer carton. The two can be separated by hand or with a butter knife. If you use a sharp knife, be careful to only separate the boxes and not cut into the boxes. If you jab into the box you might cut some of the wires for ADAM, though this isn't likely. Take the pieces out one by one, starting from the top, and place them carefully on the floor or on a large table. Once all the pieces are laid out, open the instruction packet and look through the setup manual. On pages 8 and 9 of the manual, Coleco lists all the parts that it supplies. Check that everything is included in your package.

Except for playing video games, we found it uncomfortable to have the keyboard on the floor. We were most comfortable with the keyboard on a table or desk. Keep experimenting until you find what's comfortable for you. Try putting the television on the same table or, if the set is too large, try it with the table in front of the set. ADAM's letter quality printer is fairly loud. To cut the noise a little, you can place the printer on the rug on the floor. A rubber mat might also help. Actually, one of the nicest features of any good word processing typewriter is that you don't have to be in the room while it prints, but it's best to try to cut down on the noise anyway.

Before plugging in and turning on ADAM, you might want to glance at the following description. We'll also mention some precautions worth taking as well as give some hints that will make ADAM easier and more comfortable to use.

The Components of ADAM

The ADAM computer system consists of three main pieces of equipment (the printer, the keyboard, and the system unit), two game paddles, and some connecting wires. Three cassettes, a switch, a plug adapter, and instructions also come with the system. One major part of the system is not included; that's your television set. The components are connected by wires—in most cases, many wires bound together as a cable—as shown in Figure 1.1.

The whole system, except the television, receives its power through the printer. The printer contains a power supply which converts the electricity

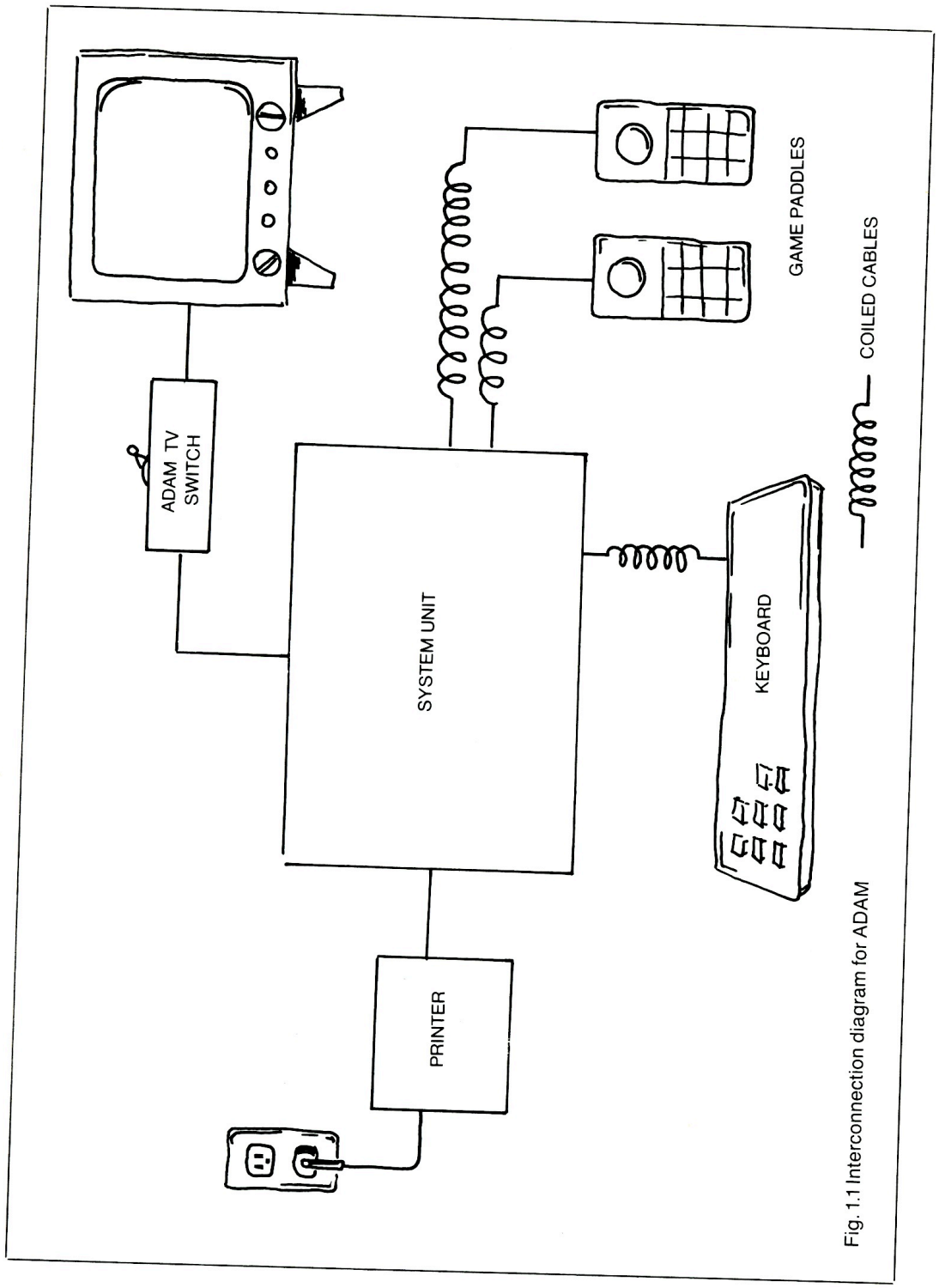


Fig. 1.1 Interconnection diagram for ADAM

from the wall plug (110 volts) to the kind needed by the ADAM system. Let's start with the printer.

The Printer

The two relatively new types of printers used now are *dot matrix* printers and *daisy wheel* printers. Dot matrix printers are much faster, but, except for very expensive ones, the print quality is never as good as with daisy wheels or even an old fashioned manual typewriter.

The printer on the ADAM is a daisy wheel printer. Not too long ago, all typewriters had long hammers. When you pressed down on a key, the hammer for the letter you hit moved up to strike an inked ribbon. You usually had to press hard on the keys to get nicely printed letters. Later typewriters, like the IBM Selectric, use little "golf balls" with letters on them. When you press a key, the ball moves around until the letter you pressed is lined up with the ribbon. Then the ball slams into the ribbon and writes the letter on the page. Golf ball typewriters are fairly slow and have too many moving parts. Moving parts break much more often than electronic parts.

To see the differences between the way each of these print, look at your utility bill. This was probably printed on a dot matrix printer. What comes out of ADAM's daisy wheel printer is much more pleasing to the eye. ADAM's daisy wheel printer works much like any daisy wheel printer. It is a simpler version of the golf ball idea. All the symbols (letters) that you want to print lie at the end of little spokes attached to a wheel. (This is the reason it is called a daisy wheel printer—the spokes look like petals on a flower.) When you hit a key, three things happen. Say you hit the key for *A*. First, the wheel spins around until the letter *A* is on top of the wheel. Second, a hammer pushes this petal into the ribbon and the *A* appears on the paper. But the printer is not done yet. Finally the whole assembly, containing the wheel, ribbon, and hammer, moves to the right exactly one space. Now the printer is ready to print the next letter. This is exactly what happens when you use ADAM as an electronic typewriter. You can see the hammer actually striking the petal.

There are several other ways to have ADAM's printer print. We'll see those later.

One of the biggest advantages of ADAM's printer is that it is easy to change the daisy wheels. This is explained completely in the setup manual that comes with ADAM. Be careful when changing wheels that the new wheel being installed does not get caught under the clear plastic paper

guide that curves around the paper roller. This roller is called a *platen*. One nice feature of the ADAM printer is that many standard ninety-six-petal plastic daisy wheels may be used. There are literally dozens of different daisy wheels that will work in the ADAM: for example, some plastic Diablo wheels fit in the ADAM. Make sure that the wheel you purchase is a 10-point size. (Ten point refers to a certain size letter.) You can type in Greek, French, Scandanavian, Hebrew, or Japanese if you can find the right wheel—and these wheels can be found with just a little effort. Other types of wheels give you scientific symbols and many different styles of print, called *fonts*.

You can write or call Coleco to see whether they have a particular wheel you want. Another good place to look for these daisy wheels is a large office supply store or mail order office supply house specializing in computer oriented supplies. The cost isn't much; most daisy wheels can be bought for less than ten dollars. For the first year or so that ADAM is available, the catalogs may not list which wheels fit on your printer. For this reason, it might be a good idea to take the wheel from the printer with you to a store to match up the size, the small rectangular notch, and the position of the characters on the petal.

> REMEMBER THAT ADAM'S PRINTER USES ONLY NINETY-SIX-CHARACTER PLASTIC DAISY WHEELS.

There are several different types of paper which can be used in the ADAM printer. Individual sheets of paper are fine for typing a short letter. To print anything longer using the word processor, you can use the Smart Keys to make ADAM stop after printing each page. This lets you insert another sheet of paper and then you can have ADAM continue printing. We prefer to use a type of paper called *fanfold* paper. This is a very long piece of paper that is folded into standard size sheets. It comes apart easily at the folds, forming regular size sheets of paper.

Most fanfold paper has little holes on thin throwaway bands along the sides of the page. The holes fit into special wheels—called *tractor feeds*—which keep the paper from sliding around in the printer. Coleco says that it will sell an inexpensive tractor feed for the printer. (You can decide for yourself whether the announced price of $125 is inexpensive.) Using a tractor feed with fanfold paper is a good idea. Otherwise you can use fanfold paper without the tractor feed bands—just tear them off.

Fanfold paper is available from office supply stores, your local Radio Shack store, and computer stores.

When you insert paper into the Smartwriter printer, it often gets caught under the clear plastic paper guide. If you continue turning the platen, the paper will get crumpled. The problem is easy to solve. Place an index card (or a *thin* piece of cardboard) between the paper guide and the platen. As you turn the platen, make certain that the paper stays on the platen side of the index card. (Some of the Coleco engineers refer to an index card used this way as a *shoehorn.*)

The Keyboard

Whether you are doing word processing or computing, the keyboard is the usual place from which to send information to ADAM. The ADAM keyboard is just like a standard typewriter keyboard with a few extra keys. The keyboard is exceptionally good for a nonprofessional machine. If you spend a lot of time at the keyboard it's important that it have a good "feel." We think the ADAM keyboard does.

The layout of the main typewriter keys is the traditional QWERTY pattern, shown in Figure 1.2.

The name QWERTY may seem ridiculous, but those are the letters on the keys above your left hand. The QWERTY (pronounced kwerty) layout is about a hundred years old and almost everyone who uses a typewriter is comfortable with it.

There are a few extra characters on the ADAM keyboard like ^,~,\,¦. The remaining keys are also different from typewriter keys.

When you first turn on the ADAM, it works like an ordinary electric typewriter. If you press the key marked WP, you turn ADAM into a word processor (we'll describe the word processor later on in this chapter). With two exceptions, all the remaining keys do special jobs when ADAM is a word processor. When ADAM is used as a video game, all of the action is controlled by the game paddles and the keyboard is not used. When ADAM is used as a computer, the key marked CONTROL is the only special key that you usually need. And then there is the mystery key called WILD CARD. It doesn't do anything right now but we think that Coleco placed it on the keyboard for two reasons. First, there is room for another key there. The second reason is not so clear. The engineers who design a computer don't always know the uses for the machine that will develop. So they leave room for extra things. The wild card may be for one of these new things. It's possible that Coleco has a use in mind for the key but doesn't want to say what it is yet. We'll have to wait to find out.

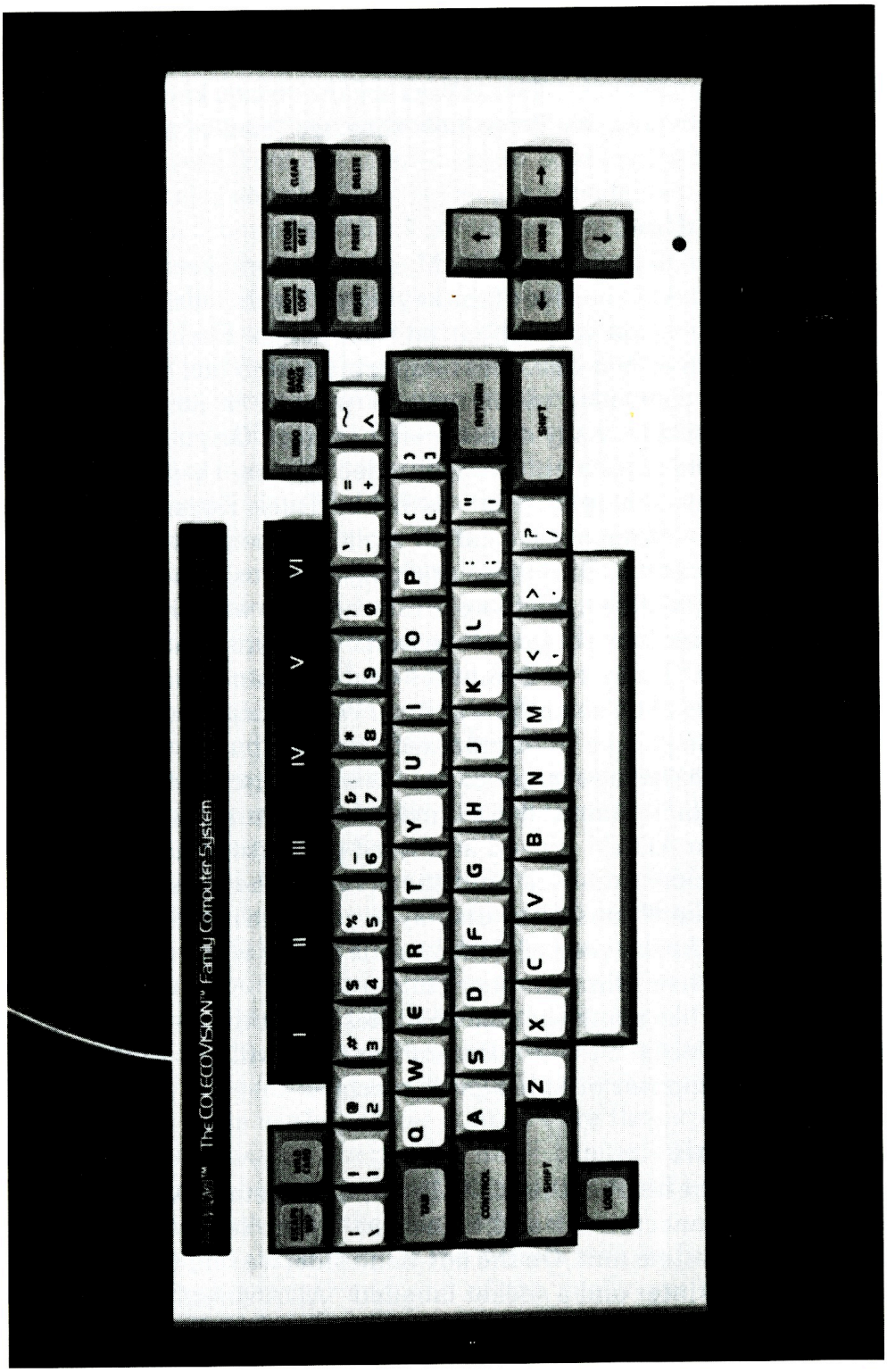


Fig. 1.2 Picture of the keyboard

When ADAM is used as a computer, all the typewriter keys are used. The special keys that are used are the CONTROL and five other keys which are used for editing programs. These keys are the HOME key and the four keys marked with arrows. We'll see how they are used in the Interlude beginning on page 79.

The Game Paddles

Two game paddles come with the ADAM, as does a bracket for mounting one of the two paddles to the side of the keyboard. Each paddle has a small keyboard like those found on pushbutton telephones. Each also has two pushbuttons, one on either side. Most important, each has a joystick.

A joystick is a controller which may be pushed in any direction—forward, back, sideways, or any direction in between. Originally joysticks were used to control airplanes. Now, in video games, they are used to change the position of a spaceship—as in the Buck Rogers game that comes with ADAM—or a mouse, a car, or a plane. The joystick has many more uses. We can use it to draw lines or change the position of objects on the screen. In short, it allows you to work interactively with the screen display. *Interactive* is another computerese word. It means that we can take actions to change the way the computer functions and the computer's actions may cause us to do something in turn. We can also use the two side buttons on the game paddle to interact with the computer.

The game paddles have yet another use. As we mentioned before, one of the game paddles can be mounted in a bracket which attaches to the side of the keyboard. When ADAM is used as a word processor, the keys on the game paddle can enter numbers onto the screen. The symbol * gives a decimal point and the # causes the display to go to the next line—in computerese, # *forces the line feed.* This feature is handy when typing lots of numbers, such as when filling numbers into a table.

Mostly, what we like about the game paddles is that the designers made them useful for each of the three different ways you will use ADAM - as a word processor, computer or video game machine.

The System Unit

The system unit is the heart of the ADAM computer system. All other parts of the system are connected to it and, in fact, are controlled by it. Let's take a short tour of the system unit. On the left-hand side, we find the socket for the wire from the printer and a socket labelled "ADAMnet," which is not yet used. Just what is ADAMnet?

Computers are sociable; they like to talk to one another on the telephone. It takes a special adapter called a *modem* for computers to be able to use a telephone. This is because telephones were designed for human voices long before computers were ever built. Modems convert the electrical pulses from the computer to tones similar to those used in pushbutton telephones. These tones are transmitted over the telephone lines and then reconverted back into electrical pulses at the other end by another modem. There are lots of reasons that we might like our computers to be able to talk to one another. Some people have used giant data banks to find people to talk with; they become electronic pen pals. Other data banks are open to anyone willing to pay for their use. For example, the Dow Jones News Retrieval Service provides access to many forms of business-related information including up-to-the-minute stock market reports. We can order consumer goods by telephone using the computer and pay some of our bills the same way. The French telephone company decided that it is cheaper to supply every home with a small computer than to have telephone directories and directory assistance. Directory assistance in France will soon be obtained by having your computer talk to the one at the telephone company. (Some people, especially teenagers, have been using their home computers to break into large computer systems. While this may seem like great fun, it is not only illegal but also very dangerous. For example, some medical records are kept in data banks so that doctors can have immediate access to them using their computers and modems. Playing with these records could have serious, even deadly, consequences to people in emergency situations.)

The ADAMnet socket is the outlet from the ADAM into the telephone lines. If you want to use it you must buy the modem.

On the back of the system unit are various sockets used to connect the system unit to your television set or to some other type of display unit. More about that later. On the right-hand side there are connectors for the game paddles and some gold electrical contacts on a board. These contacts will probably be used to connect additional things to ADAM in order to make it even more useful. These contacts are the expansion interface for the ADAM. *Expansion interface* is more computerese. It means a place where you can add on more things. Exactly what will be available to add on depends on Coleco and other manufacturers.

When we look at the fron. of the system unit we find the socket for the cord from the keyboard and one of the really exciting features of the ADAM—the digital data pack drive. There is actually one drive in place

and a cutout for another drive. The data pack cassette appears similar to the cassettes you use in an audio cassette recorder, but contains much higher quality, computer grade tape. The data packs don't fit in ordinary cassette recorders and ordinary tape cassettes don't fit in the data pack drives.

> DON'T EVER TRY TO FORCE THE WRONG TYPE OF CASSETTE INTO A CASSETTE RECORDER OR DATA PACK DRIVE.

The holes in the plastic case of the cassette and data pack are in different places. At best, you'll break the cassette case and the fragments can ruin the tape head or heads. At worst you'll completely wreck the drive. Figure 1.3 shows the data pack and an audio cassette.

Most inexpensive computers use cassette recorders for storing information. Don't be fooled into thinking that the data pack drive is an advertising gimmick for a standard cassette recorder. With cassette recorders, it takes forever to transfer information from a tape into the computer. The tape has to be wound or rewound by hand to find whatever program or data you wanted to move into the computer. Not so with ADAM's data pack. It transfers information almost as fast as an inexpensive floppy disk drive. It also has a controller that automatically lines up the tape to the information you want to transfer and then does the transferring. To us, the most innovative piece of design work in the ADAM is in the data pack drives. The drives set the ADAM apart from most other home computers. We'll discuss how to use these drives in more detail in the next section.

You may have noticed that we haven't mentioned the part of ADAM where the computer lives. We've discussed all sorts of things that are or can be connected to the computer. These things are called *peripheral devices*, meaning that they lie on the fringes of the computer. The computer itself is explained in chapter 2, but as long as we're exploring the system unit we might as well find the computer.

On top of the system unit there is a rectangular piece of plastic that can be lifted off by placing a finger in the cutout along its edge and lifting up. *Don't touch anything inside*! You can see the back of the data drive. More importantly, you can see three connectors. They are the pairs of rows of gold pins. They are mounted on a board with thin silver-looking wires on the surface of the board. The slots will be used to expand the ADAM, but it is the board that is of most interest to us. This board is called the *motherboard* and covers most of the bottom of the system unit. On the mother-

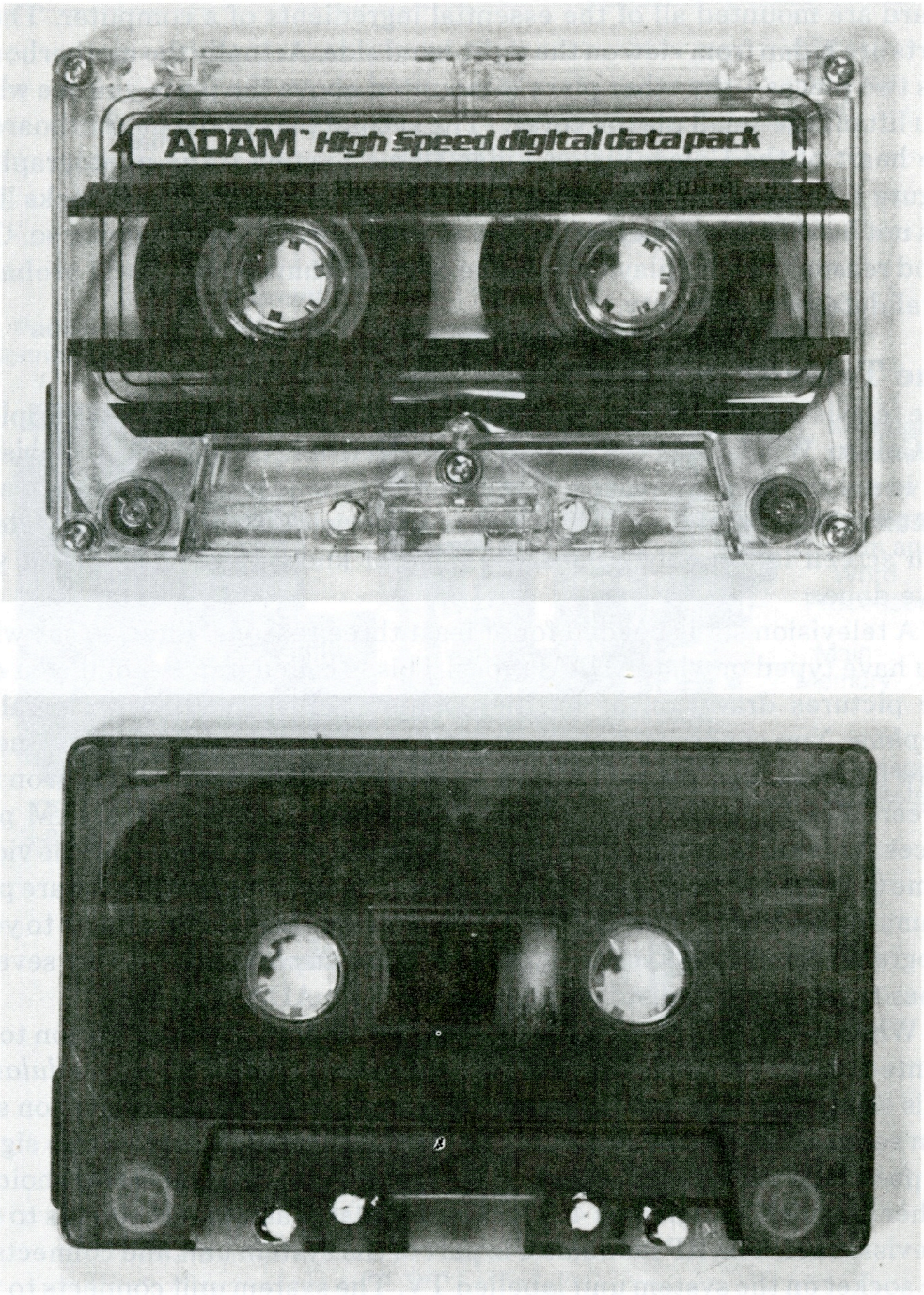


Fig. 1.3 Data pack and audio cassette

board are mounted all of the essential ingredients of a computer. These parts are hiden from view on the right-hand side. Actually the motherboard has two pieces—the other piece is mounted above the piece you see when you lift off the small plastic cover. (The second part of the motherboard is the heart of the ColecoVision Video Game System.) The photograph in Figure 1.4 should give you some idea of what the motherboard looks like. It's not hard to open the case but we strongly advise against doing so. One good reason to avoid playing with the open machine is that it will probably invalidate your warranty.

The Television Set

The television set is the only part of the ADAM system that is not supplied by Coleco. When using the ADAM as an electronic typewriter, a television set is not necessary because whatever you type is printed automatically. For all other uses of the ADAM system a television is essential. The television screen is the only means you have of knowing precisely what you have done.

A television set is needed for at least three reasons. First, to see what you have typed or what ADAM types. This is called *text.* Second, you can see pictures drawn on or by the computer. Picture drawing is called *graphics.* You can write a program, use one from this book, or buy one to draw pictures. Video games are just special types of pictures drawn on the screen. Finally, the television set lets you hear sounds that ADAM produces. You can hear some of the possible sound effects by playing the video game (see the section on Using Your ADAM in this chapter).There are programs in the Afterword that let you play music or add sound effects to your programs. A television screen is a type of *video display* and there are several other types of video displays you can use with ADAM.

When you use a television set as a video display, the information to be printed on the screen is first sent to a translator, called an *RF modulator.* This sends the information to your televison set, much as a television station sends out its signal. You can choose the channel to receive this signal by pushing a switch on the back of the system unit. You have two choices: either channel 3 or channel 4. The RF modulator also sends sounds to the television set. The RF modulator is part of the system unit and connects to the socket on the system unit labelled TV. The system unit connects to the television set using a shielded cable, like those used to connect a cassette deck to a stereo receiver. The cable plugs into a switch box that has a short wire which connects to the VHF antenna terminals on your television.

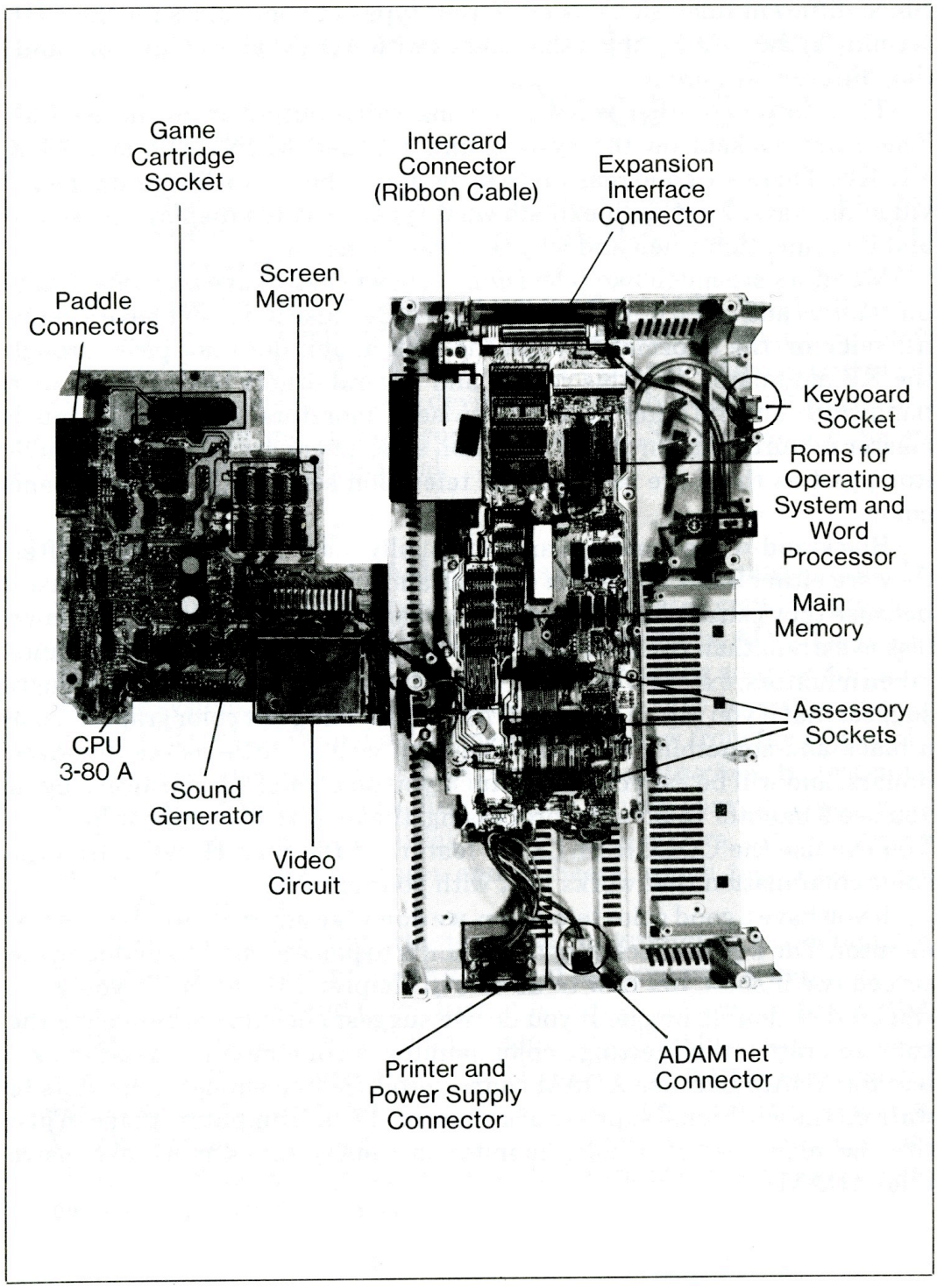


Fig. 1.4 Inside of the System Unit

Since different televisions use different types of connectors for the VHF terminals, the setup manual that comes with ADAM gives hints for handling different situations.

There are two other ways of getting video output from the ADAM. There are sockets on the system unit labelled MONITOR and AUX VIDEO. There are good reasons to know about these two different types of video displays. First we'll explain what types of video displays these are and then mention when and why they can be useful.

Monitors are much like televisions. They usually have only one (built-in) *channel* and often don't have a speaker. Because it doesn't have a channel selector, the information from the system unit does not pass through the RF modulator. For this reason and several others having to do with bandwidth and filtering in television sets, monitors can produce much clearer pictures than ordinary televison sets. There are two types of monitors, just as there are two types of television sets: black-and-white and color.

Black-and-white monitors are not usually black and white; more often they are either black and green or black and amber. These colors are used because they cause less eye strain then black-and-white screens and much less eyestrain than color screens. In writing this book we used black-and-green monitors, except when using ADAM's color capabilities. We strongly suggest that if you regularly use ADAM for anything but color graphics, use a black-and-something monitor. They can cost as little as one hundred dollars, and will be especially helpful if you do a lot of word processing. If you use a monitor to do word processing, the picture may appear blurred. You can use the COLOR SELECT feature of the SmartKeys to find the color combination that works best with your monitor.

If you have a good color television, we don't see any reason to buy a color monitor. The way to check your television is to place a small color dot on the screen (we'll show you how to do this in chapter 14) and see if you get a smeared or double image. If you do, we suggest checking or replacing the color set, rather than getting a color monitor. A color monitor is rather useless for ADAM because ADAM cannot produce fine enough color dots to warrant the additional expense of a monitor. Those computers that can justify the high cost of a color monitor are many times more expensive than ADAM.

Hints and Precautions

This is the most important section in the whole book. We wrote it to give you the hints and cautions you will need to use ADAM for many years. The cautions will tell you how to use the machine so that the safety of both you and the system are ensured. Some of the hints are fairly impractical; you can follow them or not depending on your pocketbook and your concern for the well-being of ADAM. We will try to indicate the approximate cost of any safety devices we suggest. These devices can usually be bought at a nearby Radio Shack store and at most computer stores. Several other hints are given in the ADAM setup manual.

1. Use a grounded outlet.
   The power plug for the ADAM system is a three-prong plug and should, whenever possible, be plugged into an outlet that accepts this type of plug. If you must use an outlet with only two contacts, ADAM may still be plugged in by using the three-prong-to-two-prong adaptor supplied with the system. The installation manual shows how to fasten the adaptor to the wall socket. Proper attachment of the screw from the adaptor to the socket is very important. *Don't ignore this instruction*! By fastening the adaptor to the socket as directed, you usually convert a two-prong outlet into a grounded outlet. (There are testers, which can be purchased at hardware stores and at Sears, which test whether an outlet is grounded.)

   There are several important reasons for grounding any computer. Most important is safety—both yours and the computer's. In the wintertime, when houses tend to be drier, it is easy to get a small shock after walking across a rug and touching a doorknob. That little shock is often 2,000 volts! It isn't dangerous to us. Mostly its a nuisance; but it can *fry* the delicate electronics inside computers, stereos, and televisions. Computers are the most vulnerable to these static electrical discharges. If your computer is grounded, the static electricity is usually carried away from the machine.

Another important reason for grounding the system is that grounding cuts down the interference with other television sets caused by all microcomputers. Coleco's manuals deny responsibility for any interference that your computer may cause to other people's reception of commercial television stations. The Federal Communications Commission has certified that ADAM may be used in residential areas (it's nice to know that home computers can be used in residential areas) but you, as owner, are responsible for making sure that your neighbors can watch football while you operate ADAM. We don't think the problem is as bad as the disclaimers would have you believe, but if you find your neighbors up in arms every time you use ADAM, try some of the following tricks.

2. If ADAM is causing television reception problems . . .

   a) Move ADAM. A move of twenty feet can totally change horrible reception on nearby televisions to beautiful reception.

   b) Plug ADAM into a different outlet. Sometimes computers broadcast their interference over the power lines. Changing outlets may change the power line you are on.

   c) Buy a line filter. Line filters cut interference on power lines. If your neighbor plugs a line filter into the wall and plugs his television set into the line filter, interference through the power lines should be eliminated. Line filters also work when you plug ADAM into them. The costs of line filters vary widely, starting at about twelve dollars.

   d) Buy coaxial antenna wire for your neighbor. Coaxial (round) antenna wire picks up less interference than the flat antenna wire. Ask the dealer who sells you the wire to explain how to install it.

   e) Call the FCC and ask for help.

   If you are one of the few people who have this problem, we hope that these hints prove useful, but, like Coleco, we disclaim any responsibility for them working.

3. Buy a surge protector.
   If you live in an area where electrical power lines are above ground, you might want to buy a surge protector. This will cut down on the

possibility that lightning or other sources of power disturbances might wreck ADAM. The protectors can be bought at computer stores or Radio Shack. The prices start at about forty dollars. They are often combined with a line filter.

4. Buy computer insurance.
   Instead of a surge protector you could purchase computer insurance. Most home insurance policies don't cover lightning or other surge damage, so you may have to get special insurance for computers. First check with your insurance broker to see whether your computer is already covered by your homeowner's insurance.

5. Keep food, liquids, dust, and smoke away from ADAM.
   Liquid spilled on any part of ADAM could cause you to get a serious shock and will almost certainly destroy all or part of the system. The moisture in foods can cause the same problems as liquids. Smoke or fine particles of dust can destroy the information on the data packs.

6. Keep the system unit and printer adequately ventilated.
   These parts give off a fair amount of heat. Without sufficient ventilation, the electronic parts in them will cook.

7. Keep your hands away from the electronic contacts inside the printer and system unit. Don't touch the contacts on the right-hand side of the system unit.

8. Don't touch the tape in the digital data pack.
   The oils on your skin are enough to destroy the information on the tape.

9. Keep your fingers away from the bottom of the data pack drives.
   The tape heads in the data pack drives are also sensitive to the oils on your skin. In addition, the positioning of the heads is critical and, if they are pushed hard, their position will change.

10. Clean and demagnetize the data pack drive heads.
    Like most tape heads, the heads on the data pack drives will eventually pick up little metallic pieces that flake off the tapes. These pieces should be cleaned off. After several months of intensive use, we did not

find a significant accumulation of these particles, so cleaning was not yet necessary. After a lot of use, the heads might become magnetized. If you regularly get error messages from ADAM saying I/O ERROR, then you might suspect that the heads are dirty or have become magnetized. Coleco has not yet announced an approved method for demagnetizing but they have a kit containing a head cleaner. Call them to get the kit and find out what they suggest about demagnetizing. We used the standard method for tape recorders, but we do not recommend that you use it before checking with Coleco.

11. Don't turn ADAM on or off with a data pack in the drive or on the printer or system unit.

12. Never place a data pack on the printer.
    When you turn electrical devices on or off, a magnetic field is created that could be harmful to the information on the data pack. The power supply in the printer always creates a magnetic field. Keep the data packs away from any device containing a motor.

13. Never open the data pack drive door while the tape is moving.
    Opening the data pack drive door may cause the data pack to fly off the drive. This results in something on the floor that looks like brown spaghetti.

14. If your picture is sloppy, buy a shorter wire to connect the system unit to the television.
    A wire can act as an antenna and cause a really sloppy picture. Shorter wires can be bought at Radio Shack or other electronics supply stores. Bring the long wire with you to make sure you get the right kind of plug. A short wire costs a few dollars.

15. What to do about overscan.
    Most television sets are adjusted so that the picture does not fit on the screen. This is called *overscan.* Usually you don't notice this because television stations don't put important parts of the picture along the edges. Unfortunately, most home computers use the whole picture. You may find that characters you type simply fall off the edge of the screen and cannot be read. If you're lucky your television has an adjustment called *horizontal width.* By changing the setting on this control,

you can make the picture fit on the screen. The usual broadcast picture may not fill up the whole screen after you change the setting, but at least you can control the picture somewhat.

If you don't have a horizontal width control, you can either have a television repairer change the width setting or hope that Coleco has found a solution to the problem—call them.

Using Your ADAM

Now that you are familiar with all the separate pieces of the ADAM system, it's a good time to connect it all together and plug it in. The assembly directions that come with the system are quite clear and will lead you step by step throught the assembly process. The only tool you need is a screwdriver to connect the whole system to your television set. Then turn it on.

When ADAM is first turned on, it functions as an electronic typewriter. Whatever you type on the keyboard is typed on the printer. If you don't know how to type, there are many books, and soon there will be computer programs, that will teach you to type using ADAM.

Video Games

The ADAM is equipped to use ColecoVision Video Game cartridges. ***Be sure the ADAM is turned off before inserting a cartridge***. You are now set to play the video game using the game paddles. When the game is over it will usually give directions to restart it. If not, you can always restart it using the right-hand sliding switch on the top of the system unit marked CARTRIDGE RESET.

There are also video games which come on digital data packs. Buck Rogers' Planet of Zoom is included with the ADAM system. To use these video games, first turn the system on. It comes up (***comes up*** is computerese for what a computer does when you first turn it on) as an electric typewriter. First open the door to the data pack drive by gently pushing the release button on the system unit which lies directly above the data pack drive. Now carefully place the digital data pack for the video game into the data pack drive. The label on the pack should face you. Gently close the door to the drive — ***do not force the door shut***. If the door doesn't close easily, then the data pack is not in the right position. Move it around and try to close the door again. After a while you'll find this very easy. Push the left-

hand sliding switch marked COMPUTER RESET. The game gives you all the instructions you need to play using the game paddles. We've never been video game fanatics, but we enjoyed playing Buck Rogers long enough to delay the writing of this book for several days.

Word Processing

Another use for the ADAM is word processing. Actually ADAM seems to have been designed very much with word processing in mind since a high-quality daisy wheel printer comes with the system.

What exactly is word processing? A word processor is the ultimate fantasy machine for a frustrated (or bad) typist. When typing on ordinary typewriters, which isn't often anymore, we sometimes want to change a word that appeared ten lines up. But it's already on paper so it can't be changed without erasing and retyping and possibly having to move words to the next line. That's really a mess. Or sometimes we decide that a particular sentence doesn't belong in the first paragraph but rather in the tenth. Wouldn't it be nice to move it by touching a few keys? Or perhaps we want to send a letter to our congressman complaining about something. The first draft is likely to contain language that would make a sailor blush. Maybe tomorrow, after a few changes, we can produce a decent letter. But we really don't want to retype all of the pages again. Wouldn't it be nice to just make the changes and then push a button which will have the whole letter printed? A word processor does all those things and more.

This book was written on a word processor which is quite a bit more expensive than ADAM (five times as expensive to be precise). ADAM is not really set up for book-length manuscripts. We also needed telephone communications and floppy disk drives, which ADAM did not yet have. But all the features of a word processor that we need are in the ADAM. Almost any computer can be turned into a word processor by adding a printer and a word processing package (*package* is computerese for a collection of programs). The least expensive of these add-ons, that are worth buying, cost about the same amount as an ADAM, and you still need to buy a computer.

We don't intend to spend much time describing how to use ADAM as a word processor. For one thing, the video display tells which of the SmartKeys—those are the large black keys on the keyboard with Roman numerals on them—to use to do almost anything you might want. The six black keys in the upper right of the keyboard have labels that are almost self-explanatory. You can begin word processing without opening the

manual or using the Easy Reference Guide—both of which come with ADAM. The manual for the word processor is quite readable.

We wrote this book to help you teach yourself to write computer programs for the ADAM using SmartBASIC. SmartBASIC is similar to the version of BASIC used on the Apple computers. But the differences are sometimes fatal. Coleco advises that any Applesoft program will run on the ADAM. This statement, at the time of the writing of this book, is not completely correct. We will tell you why after we describe the working of ADAM's computer in chapter 2. For now, we can expect that Applesoft programs will be the ones most often used on ADAM. This will continue to be true until people start writing programs making them available either by selling them or publishing them in places like the *Family Computing Magazine* published by Scholastic, Inc.

Up and Coming for ADAM

We first saw the ADAM computer system at its second public demonstration. We liked the features of the system. As we worked with ADAM to develop this book, we decided that it is a most impressive computer. It is interesting that the president of Coleco once described ADAM as a typewriter that plays "Donkey Kong." Either he was joking (and we're not really sure) or he doesn't comprehend the quality of the product that his engineers have designed. ADAM, like other computers, had many early production problems. This is typical in the computer industry but is not common in the toy industry. For example, there were serious problems with the IBM PC when it was first released, but no one remembers that now. The PC was never a state-of-the-art machine—it was conservatively designed—and also costs many times more than ADAM. In the computer business there are ways to avoid production problems, but they can double or triple the price of the product. For example, the best way to test a computer is to run it for about one hundred hours and then fix whatever doesn't work. Even car manufacturers can't do this and cars have a few thousand parts. ADAM contains in its few parts about one million transistors; if one breaks, the computer won't work. It usually takes about six months to set up good testing procedures and, after the tests are developed, the com-

puters will usually be reliable. At any rate, the flaws are minor and can be quickly fixed. To demonstrate their faith in ADAM, Coleco now offers a six-month warranty on the machine. This is unusual in the industry—for example, the IBM PC has only a three month warranty. Usually a computer that works well for three months will work well for five years (this doesn't include moving parts like the printer end and the digital data pack—they can wear out depending on how much you use them).

Once we saw the capabilities of ADAM, it was clear to us that it is one of the best computers around—at any price. When we did some speed checks in SmartBASIC, we found ADAM to be considerably faster than the IBM PC for most operations (those not using the digital data packs). The Coleco floppy disk system solves even these speed problems. Other additions can make ADAM as capable and versatile as a business computer costing several thousand dollars more than ADAM.

There are features of ADAM that we don't like. For example, the printer is too noisy. But someone will eventually design an inexpensive sound-proofing cover for it. We would also prefer to have the power supply located in the system unit rather than in the printer—if the printer breaks down, the whole computer can't work while it is being repaired.

The ADAM computer system comes with several peripheral devices and several internal devices that you never see but which control its performance. It also includes SmartBASIC and one video game. The following additional items should be available by the time you read this.

1. A modem - to allow you to connect ADAM to other computers through the telephone network. This will be under $100.

2. A 64k memory expander (see chapter 2) - this expands the capability of ADAM and would let it handle most of the sophisticated programs around. The memory expander should be used in conjunction with many of the following items:

3. A floppy disk drive - a disk drive allows programs and data to be placed into ADAM's memory quickly - five to ten times faster than using the digital data pack drives. This is a real advantage in advanced programming or in using specialized programs for business or education. Disk drives are used int he same way as digital data pack drives. The price for one drive will be about $250. A second disk drive can easily be added.

4. A second data pack drive - this comes as no surprise since there is already a cut-out for it on the front panel of the system unit. The expected price is under $200. We believe that the disk drives are much more useful and so a better buy.

5. CP/M - A versionof CP/M (a master program for microcomputers) will be available. This will be sold either with the disk drive system or for use with the digital data packs. CP/M is useful because most programs using CP/M can run on any CP/M system. There are now thousands of programs available using CP/M - educational, game, and business programs in addition to many other computer languages. With CP/M, ADAM becomes much more versatile. There are two catches - the programs that you might buy using CP/M may not be cheap and we don't know the price that Coleco will charge for their version of CP/M.

6. A tractor feed for the printer—we've already mentioned that we think very highly of tractor feeds. The announced price for ADAM's is about $125.

7. Various accessory kits containing print ribbons, daisy wheels, blank digital data packs, etc. These are standard replacement items. You might want to get a catalog of them to keep handy. The ribbons and data packs cost about $6. Coleco sells an accessory kit for around $35 that contains three daisy wheels with different styles of fonts, a spare ribbon, a data pack and a head cleaning kit for around $35.

8. LOGO is available for under $75. This is another language and is very popular for teaching. If you want to learn this very useful language we can recommend it highly. We think it one of the best versions of LOGO available at any price, on any machine. SmartLOGO (as it is called) won a prize at the 1984 consumer electronics show for its excellence.

9. Lots of educational programs. Some of these are based on the Dr. Seuss characters. There is also a typing tutor and a beginners' LOGO.

10. Programs to manage files and do spreadsheets.

We don't suggest that you run out and buy each of these options. As with any computer, adding all possible options can cost many times the original price of the machine. Some of the options will make excellent presents, others will not be useful for you. Once you get to know ADAM, you'll have a good idea of which options might interest you. We suggest being an informed consumer (and hope that you're better at it than we are—we tend to buy peripheral equipment on a whim). Based on past experience in the microcomputer buisness, we guess that many other companies will start producing programs and peripherals for ADAM. Then many new languages and programs will become available and prices may be lower. You can be fairly certain that anyone writing programs or making peripherals for ADAM wants to sell them. There will be advertisements for these products in *Family Computing Magazine.*

BUZZwords

COMPUTERESE	INTERACTIVE
EXPANSION INTERFACE	JOYSTICK
FONTS	PERIPHERAL DEVICES
FORCE A LINE FEED	QWERTY
HANDS-ON EXPERIENCE	TEXT



What is a Computer?

We showed you many of the features and parts of the ADAM computer system in the first chapter. You saw that ADAM is really a computer with several peripherals attached. *Peripherals* are extra parts attached to ADAM to allow it to do all sorts of things. In this chapter we will look at the computer part itself. We will also tell you some of the words and phrases that people use in describing computers and how they work. Strictly speaking, this chapter is not necessary to learn to write simple programs in SmartBASIC. You can jump directly to chapter 3 and return to this chapter whenever you wish.

It wasn't very long ago that computers were described as dumb machines big enough to fill several rooms. Now, computers are small dumb machines. They might be a little smarter than the monsters of twenty-five years ago and they can be as small as a hard-cover book. But they are still dumb. You might wonder why we say that computers remain stupid. After all, computers now control our cars, pay our salaries, send out social security

checks, and, in the opinion of many people, practically run our lives. But computers really can't do very much. There are only about twenty or thirty operations that a computer can perform. Moreover, these are all very basic. They are operations, like moving something from one place to another or adding two numbers together. The computer's power comes from its incredible speed and near-perfect accuracy.

We give orders to computers by writing programs and the only thing computers do is to follow those instructions exactly. The programs tell the computer the order in which they should perform their basic operations. If the program is a word processing program, and a keyboard, a video display screen, and a printer are connected to the computer, we obtain a word processor. After you dial a number, the telephone company uses computers to connect you to the telephone of the person you are trying to reach. That same computer also calculates how much to bill you for the call. In the next section we will set up an imaginary office that works just like a computer. This helps to see how a computer operates and how it communicates with the outside world.

An Office as a Model for a Computer

Imagine an office. An office should have a desk with some drawers. An extra filing cabinet would be useful and, of course, a person will be sitting at the desk. The desk has a drawer named *data drawer.* Another drawer is labelled the *program drawer* (Figure 2.1).

This office always follows fixed rules, laid out beforehand. Here are the rules.

1. All work is performed in the open, on top of the desk.

2. There is a stack of papers in the *program drawer* with instructions telling the person exactly what to do.

3. The *data drawer* starts out empty.

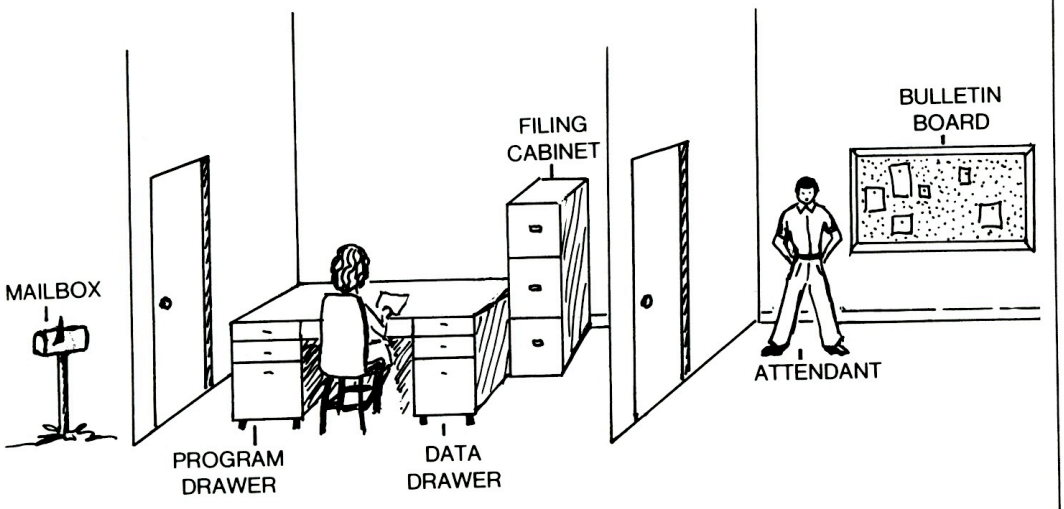


Fig. 2.1 Office

4. If we want to do anything to a piece of paper in the file cabinet, we first have to move the whole file folder containing the piece of paper into the *data drawer.*

5. We follow the instructions exactly in the order that they are stacked and only one thing is done (processed) at a time.

6. The workday starts when a bell is heard and ends when all the instructions in the *program drawer* have been successfully completed.

7. At 5:00 PM, the contents of the *data drawer* self-destruct.

There are several more rules in this office, but let's look at a typical work session first.

It's 8:00 AM. The bell rings and the worker starts by opening the *program drawer* and looking at the first instruction. It says, "Move the file folder labelled BILLS TO PAY from the filing cabinet to the *data drawer.*" The second instruction tells the worker to do the same thing to the CHECKS file folder. The third tells the worker to do the same thing to the TO BE MAILED file folder. The fourth tells the worker to move the rent bill to the top of the desk, and the fifth asks that the check for the rent bill be moved to

the top of the desk. The sixth tells the worker to place the bill and check into the TO BE MAILED folder in the *data drawer.*

All the instructions are very simple and only ask the person to do exactly one thing. The rent bill has almost been paid. So far it only sits in an outgoing mail folder. But we've obviously left out some important rules.

Leaving any file folders in the *data drawer* means that the next day they will be gone. Using two instructions, we can place the BILLS TO PAY and CHECKS folders back in the filing cabinet. However, if we place the TO BE MAILED folder in the filing cabinet, the rent won't be paid. We have to mail it. Now, no office is any good without a way out, so our office must have a door. That way we can imagine an instruction that sends for a messenger who will pick up the TO BE MAILED folder and carry it to the post office.

Of course, no one would know that the check was mailed and we probably need to have a way of letting the boss know which bills have been paid. To remedy this potential problem, we could have a bulletin board outside the office with an attendant standing by. Then we could use an instruction that would have the attendant post a sign on the bulletin board saying which bills have been paid. We can also have the attendant post questions that some instruction in the *program drawer* has listed as needing a response from outside the office. Sometimes the bulletin board is blank and a passerby can write messages on it. Unless those messages directly answer questions posted by the attendant, our office worker just ignores them.

The Inside of ADAM

The office we just constructed is a model for the inside of a computer. It isn't perfect but it is a good way to start understanding how a computer works.

The computer takes orders. The orders are the individual statements in a program. In our office the *program drawer* corresponds to the program. The file folders in the *program drawer* are the statements or instructions. The computer executes these instructions in order. (*Execute* is a word in computerese that means carrying out instructions. The special words in computerese are often called *buzzwords.*) Each instruction tells the computer to either move around some information—called *data*—or do something to a piece of data sitting on the desk.

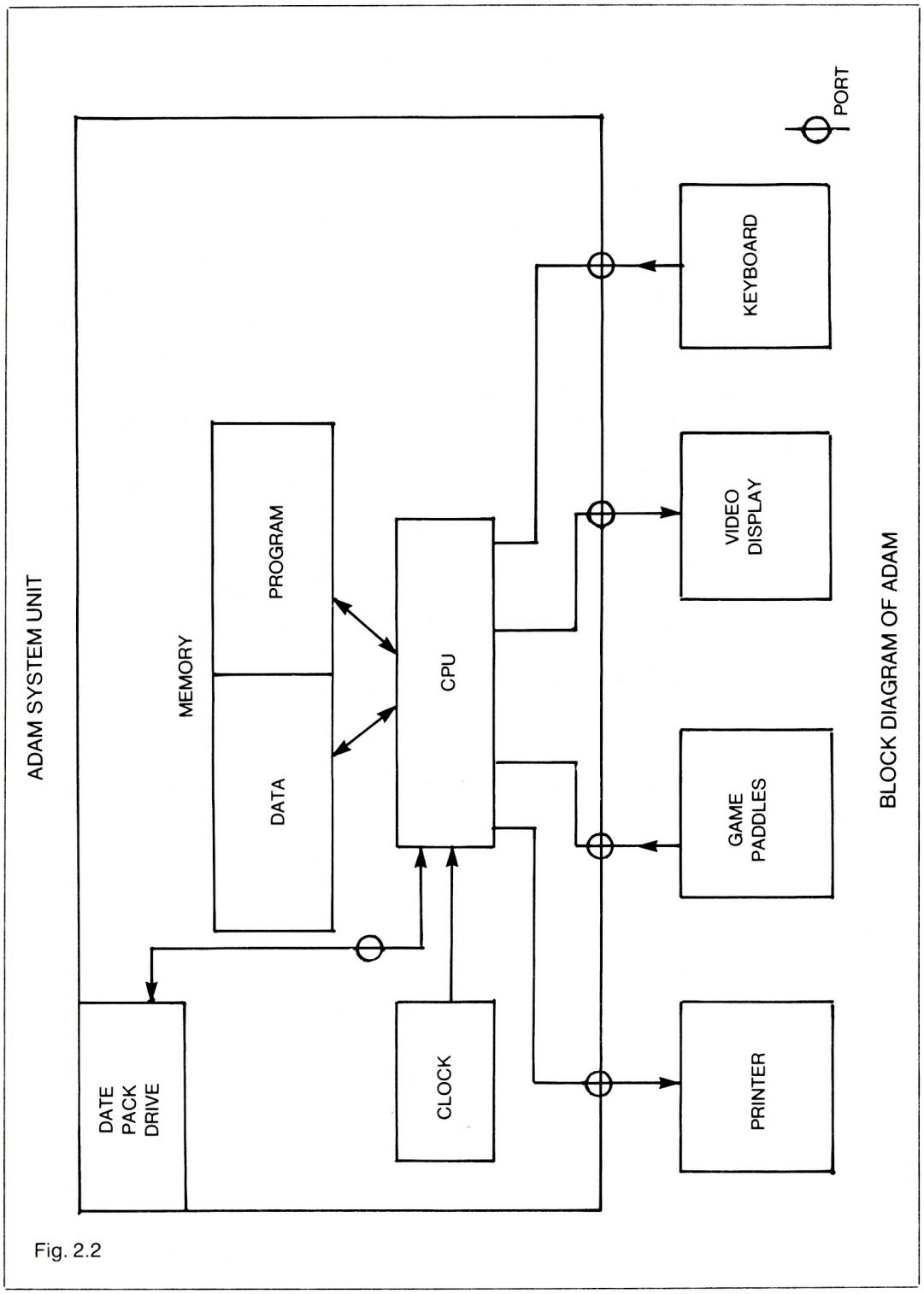

ADAM SYSTEM UNIT
MEMORY
PROGRAM
DATA
CPU
CLOCK
DATE
PACK
DRIVE
PORT
KEYBOARD
VIDEO
DISPLAY
GAME
PADDLES
PRINTER
BLOCK DIAGRAM OF ADAM

Fig. 2.2

The top of the desk corresponds to a location, called the *accumulator,* inside the computer. The filing cabinet is your digital data pack. The *data drawer* in the desk is a part of the memory of the computer called the RAM. *RAM* stands for *random access memory.* It's called random access because we can pick out any piece of information from RAM in the same amount of time. The first piece doesn't need to be in the first position (or file folder). This is very different, and works much faster, than the digital data pack. Just as was true for the information in the *data drawer,* the information in RAM is lost when the computer is turned off.

Certain programs are built into a computer. For example, ADAM's word processor and the control program for the data pack drives don't disappear when the power is turned off. These programs are kept in a special part of memory called *ROM* or *read only memory.* Think of ROM as containing the basic rules for the computer (or office). These rules never change so we don't want to rewrite them each time we turn on the power. (A computer without ROM usually needs several hours of work before it can be used.)

The doors to the office correspond to other parts of the computer. These parts are called *ports.* They form the entrances to, and exits from, the computer.

With ADAM, the keyboard and video display correspond to the bulletin board. They use some of the ports. Other ports go to the printer and the game paddles.

In our office model, all these objects—the desk, the filing cabinet, etc.—take up a lot of space. In the computer, they all sit in a little box. Even more amazing is that most of these objects are on little pieces of glass no bigger than a fingernail. The pieces of glass are called *chips* or, to be more technical, *integrated circuits.* The chips sit on the motherboard, to which they are all connected. There is a special chip called the *CPU,* or *central processing unit,* which contains the part of the computer that acts like the person and the top of the desk. It contains the accumulator, and some specialized RAM and ROM. The ROM acts like the brain of the office worker because it tells the computer how to interpret each instruction.

In the ADAM, the CPU is the Z-80A chip first made by the Zilog Corporation. It is the same CPU that is used in the Radio Shack TRS-80 and many other computers. The RAM consists of a whole block of chips. The layout of some of these chips is shown in Figure 2.3.

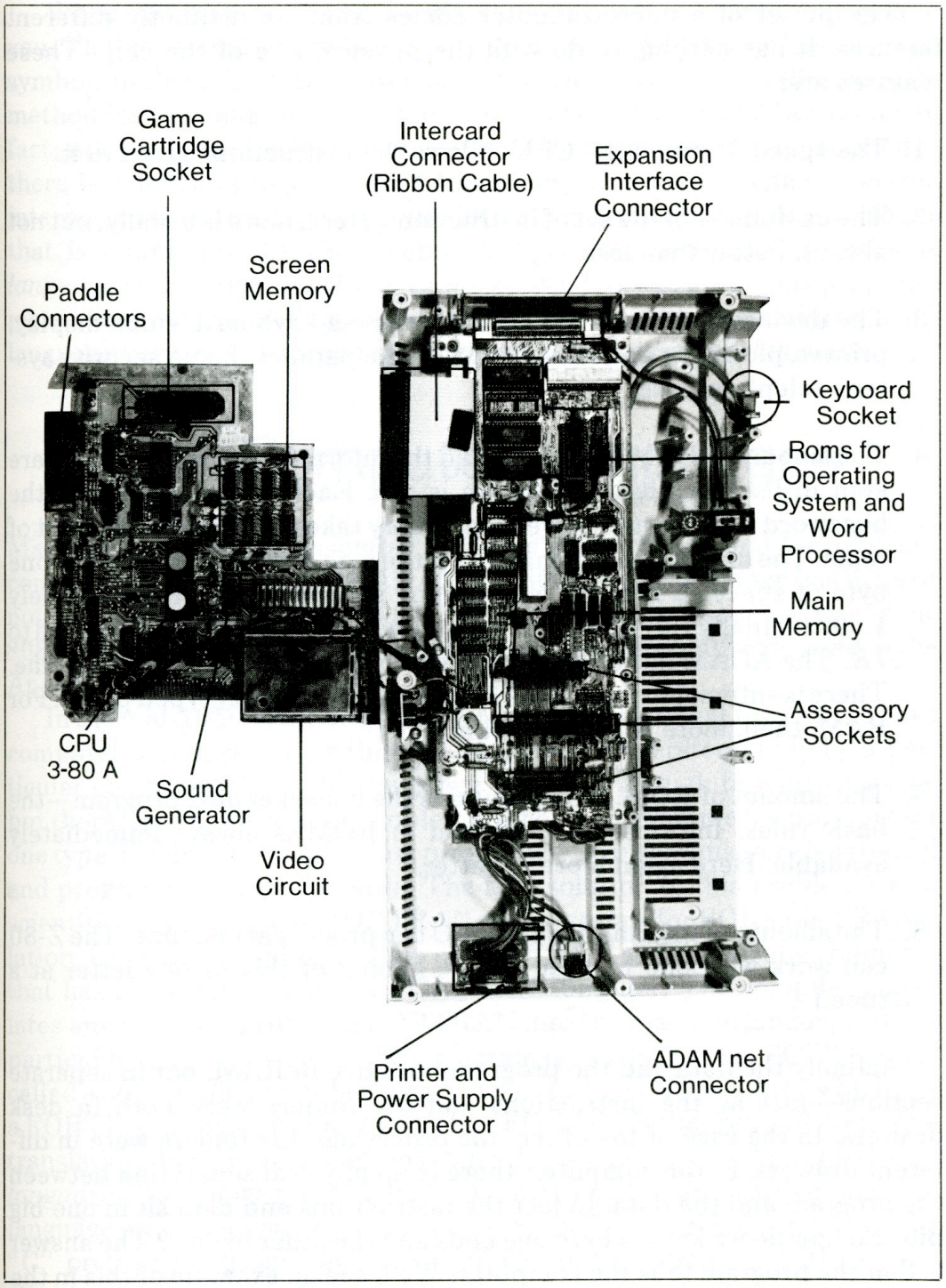


Fig. 2.3 Layout of the computer

The power of a microcomputer comes from six distinctly different features. It has nothing to do with the physical size of the chip. These features are:

1. The speed at which the CPU follows the instructions given to it.

2. The usefulness of the set of instructions. Here, more is usually, but not always, better than less.

3. The devices connected to the computer—a keyboard, video display, printer, plotter or graphics printer, game paddles, home security system, telephone, etc.

4. The amount of RAM. Programs and the information they work with are kept in RAM while the program is run. Each letter or symbol (the buzzword is *character*) stored in memory takes up a certain amount of room. The unit of space in memory is the *byte.* Each letter occupies one byte of space. A number may occupy several bytes. Approximately 1,000 (actually 1,024) bytes is called 1 *kilobyte,* which is abbreviated to *1K.* The ADAM has 80K of RAM and is thus called an 80K machine. There is sufficient space in RAM to store about thirty typed pages. For RAM, then, more is definitely better.

5. The amount of ROM. ROM contains the housekeeping program—the basic rules. Information contained in ROM is always immediately available. Here again more is better.

6. The amount of information the CPU can process at one time. The Z-80 can work with one byte at a time. (Think of this as one letter at a time.)

Actually the data and the program both live in RAM, but in separate sections—just as the instructions and file folders were both in desk drawers. In the case of the office, the orders and file folders were in different drawers. In the computer, there is no physical separation between the program and the data. In fact the instructions and data sit in one big pile. So how do we know where one ends and the other begins? The answer is that the program tells the computer! We'll see an example of this in the next chapter.

When you type on the keyboard, the computer is able to understand you. The words and symbols you type are the standard letters, words, and symbols in English. Does this mean that some engineers have discovered a method for teaching pieces of glass to understand English? Not really; in fact, not likely in the foreseeable future. Actually, inside the computer there is a special program which translates or, in more technical jargon, interprets the commands you type into commands that use the language that is understood by the computer. This language is called *machine language* and is truly miserable to work with. Even professional programmers avoid working with machine language these days. Instead they use a language very close to machine language, called *assembly language.*

Computer Languages

Machine language and assembly language are very different for different computers. For example, the ADAM's Z-80 machine language has little or nothing in common with the machine or assembly languages of either the 8088 chip used in the IBM Personal Computer or the 6502 chip used in the Apple II.

In the mid-1950's, the computer world recognized that they had to have computer languages other than the languages understood only by a particular computer. Assembly language was hard enough to learn and use, but there were more serious problems. A good programmer or program for one type of computer was useless on another. So programs were less useful and programmers less versatile. The first solution to this problem was a scientific language called FORTRAN, which stands for FORmula TRANslation. Programs could be written in FORTRAN and run on any computer that has a special program, a FORTRAN compiler. The compiler translates any program written in FORTRAN into the machine language of the particular computer on which the program is now being run. FORTRAN is called a *portable language* because it can be used on any machine which has a FORTRAN compiler. Languages like FORTRAN that require extensive translation before reaching machine languages are called *high-level languages.* Languages in which the instructions are almost like machine language are called *low-level languages.* Some low-level languages like the "C" programming language are portable and are popular with professional programmers.

2. BASIC usually works slower than other languages on most computers. For almost all home computer applications, though, the lightning speed of modern microcomputers makes this objection irrelevant.

3. BASIC does not require that programs be "structured." Structured programming is the best way known to organize large programs. Since structured programming or, more precisely, modular programming can be done in BASIC (see the discussion in chapter 12), the authors of this book don't take this objection too seriously.

In spite of these objections, for short programs whose results need to be known quickly, we prefer to use BASIC. Even for some long programs, the versatility of BASIC makes it our language of choice.

Before proceeding to programming in BASIC, two more topics should be mentioned. We have spoken before about translation programs between high-level languages, like BASIC and FORTRAN, and machine language. There are actually two different types of translators—one is an *interpreter* and the other is a *compiler*. They differ in how they translate, not in what they translate. An interpreter translates instructions one at a time, while a compiler translates a whole program into machine language before the program is used.

A compiler has two main advantages over an interpreter, both of which give programs that run faster. The first is that we can save the translation, so if we run the program again we don't have to retranslate it. We just run the translation. The second advantage occurs for instructions that are used many times. The interpreter translates the instruction each time it is used, while the compiler translates it once. If an instruction is used often, we get considerable time savings by not having to retranslate each time.

All home computers that we've seen come with a BASIC interpreter. Many have or promise to have a BASIC compiler as an extra-cost option. Programs that run using a BASIC interpreter will also run using a compatible compiler. There is almost no difference in the way the programs are written.

Before closing this chapter, we want to explain a bit about the relationships between the versions of BASIC used in the Apple computer and the one for ADAM. For more details see Appendix D.

The version of BASIC seriously used on the Apple is called Applesoft BASIC. For reasons that are mainly historical, most of the educational programs that we can now buy were first written in Applesoft. Lots of other programs were also written in this language—game programs, programs for the home and small businesses, and others. Coleco made a wise decision in giving their BASIC the ability to understand programs written in Applesoft. From the first day that you have your ADAM, you can get programs that will work on it. There is, however, a big BUT! It is reasonable to think that every program that runs on the Apple will run on ADAM. Unfortunately, many won't. The reason some won't and how to tell which will is easy. All you have to know is that BASIC allows you to write pieces of programs in machine language. Since the Apple uses the 6502 chip and the ADAM uses the Z-80A chip, they have different machine languages. Any program written for the Apple that has a part written in machine language almost certainly won't run on the ADAM. When you look at an Applesoft program and see the words PEEK, POKE, USR or CALL, you can pretty much forget about running it on the ADAM unless you do some hard work. Otherwise any program you buy or copy from a magazine should work.

Now, on to programming.

Summary

In this chapter, we showed you a way to think about a computer and took a closer look at the heart of ADAM. It is composed of electronic devices called integrated circuits or chips. The most important chip is the central processing unit (CPU) which in the ADAM is the Zilog Z-80A. Other chips are used to provide random access memory to temporarily store data (information); others provide read only memory to store housekeeping and built-in programs. Each piece of data for ADAM consists of 1 byte (or character) of information. A byte can hold one character—this could be a letter or a symbol; 1,024 bytes is a kilobyte (1K). ADAM is an 80K computer. Data moves in and out of a computer through ports. A computer follows instructions in its own language, machine language. Instructions written in ADAM's SmartBASIC, like any high-level language, must be translated into machine language before a computer can follow (or execute) them. The translator is called an interpreter or compiler.

BUZZwords

ASSEMBLY LANGUAGE

CHIPS

COMPILER

DATA

EXECUTE

HIGH- AND LOW-LEVEL LANGUAGES

INTEGRATED CIRCUITS

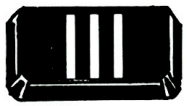
INTERPRETER

MACHINE LANGUAGE

PORTABLE LANGUAGES

PORTS

RAM (Random Access Memory)



ADAM is A Calculator

In this chapter, you will use ADAM as a calculator. To do this we will show you the important BASIC commands:

PRINT	LET
HOME	

We will also learn the symbols that ADAM uses for:

Addition	+	Subtraction	−
Multiplication	*	Division	/
Exponentiation	^		

We will be able to recognize the SmartBASIC prompt] and learn to use the *direct* or *immediate mode.*

Simple Calculations

A good way to begin programming ADAM is to first use it as a calculator. First turn on the power. The power switch is at the rear of the printer. Once the power is on and ***only then***, place the SmartBASIC tape into the tape drive with the label on the tape facing you. As we said in chapter 1, make sure the digital data pack is inserted correctly. After inserting the tape, push the reset switch on the top of the tape unit—this switch is the slide switch on the left. After a short wait a] will appear. (At this point it's a good idea to take out the SmartBASIC tape and insert the other digital data pack.) This symbol is called the BASIC *prompt* and it tells you that ADAM is waiting to take your orders. A flashing line will appear next to the]. This line is called the *cursor,* and anything you type will appear over it. It functions as a place marker.

The major difference between using hand-held calculators and using ADAM as a calculator is that with ADAM you have to use the word PRINT before the calculation. Now type:

```
PRINT 2+2
```

This is called a *print statement.* The screen shows:

```
] PRINT 2+2
```

In using ADAM, or any other computer, for that matter, touch typists run into one problem. They may have been taught that symbol 1 (one) and l (lower case L) are interchangeable as are 0 (zero) and O (upper case O). *To ADAM 1, l, 0 and O are different characters—one cannot be substituted for another.*

Now hit the RETURN key and ADAM will immediately print a 4 on the next line.

Using ADAM this way is called the *direct* or *immediate mode,* but we like to think of it as the calculator mode. (*Mode* is a buzzword meaning type of operation.) Try some other calculations using the following symbols:

+ addition
− subtraction
/ division
* multiplication (This is the only unusual symbol. Don't use an x.)

Wait for the prompt] to come onto the screen before you start typing.

Each time you type something in and hit RETURN, the prompt] moves two lines down the screen. When it reaches the twenty-fourth line it stays there. Each new line that you type in or that ADAM prints out will cause what's on the top line to disappear and everything that's displayed on your screen to move up. The buzzword for this movement is *scrolling.*

If you get tired of a cluttered screen, ADAM lets you erase everything. Type HOME and hit RETURN. This command clears the screen and places the cursor in the upper left corner.

The HOME command acts differently from the HOME key. The HOME key only moves the cursor to the upper left corner but does not clear the screen.

One thing to beware of is that ADAM can't tolerate a comma when used in numbers. If you type PRINT 1,000 + 1,000 and hit RETURN, some nonsense is printed on the screen. This is an example of another word in computerese, *gigo.* These initials stand for "garbage in, garbage out," which means that if you type in something that the computer misunderstands, you get out nonsense. If you're lucky when you type in something wrong, the computer will tell you that it can't understand your instruction. Sometimes it will even tell you why. If you type:

```
] PRINT2+2
```

ADAM will tell you that this is an ILLEGAL COMMAND. Because you didn't leave a space after PRINT, ADAM looks for the command PRINT2+2 in its dictionary, but it isn't there. PRINT is the word in the dictionary. If you get this or any other *error message,* don't worry. It just means that you typed in something that ADAM can't understand. At this point the best thing to do is just to retype the statement.

REMEMBER: NOTHING THAT YOU TYPE CAN DAMAGE ADAM.

Let's go back to using ADAM as a calculator. Most calculators have a lot more than the four basic functions of addition, subtraction, multiplication, and division. Since ADAM is much more sophisticated than any hand-held calculator, you might be wondering about some of the other functions that are built in. If you want you can turn to chapter 9 and see how to type in your favorite function. The form that you must use goes under the buzzword of permissible (or allowable) *syntax*. That's why using commas in numbers sometimes results in ADAM printing out SYNTAX ERROR. This just means that you didn't use a form that ADAM recognizes or that your "grammar" is not acceptable to ADAM. As an example, you might try:

```
] PRINT LOG (1,000)
```

Self-Check 1

A. Try the following. Type one in, then hit return. Then try another.

```
1.  PRINT 1.09/1.09
2.  PRINT 5.6 - 2.01
3.  PRINT -6 + 2
4.  PRINT 1/3
5.  PRINT .00000001/123456789
6.  CLEAR the screen
```

B. Something is wrong with each of the following examples. Try each on ADAM to see what response you get. If possible, try to correct the mistake in each and then retry them on ADAM.

```
1.  PRINT 1,000*20
2.  PINT 6*4
3.  PRINT 2 x 3
4.  PRINT 2x3
5.  PRINT 6.1/0
```

Exponentiation and E Notation

Raising a number to a power, which is also called *exponentiation*, means multiplying a number by itself a certain number of times. The symbol you may have learned in your math courses is a bit different from the symbol used by ADAM. To multiply 2 by itself 3 times you probably wrote 2^3 to mean (2) (2) (2) = 8. In SmartBASIC the symbol for exponentiation is the caret ^ which is located on the top right white key. To do this computation using ADAM we can type in PRINT2 ^ 3 and hit RETURN. ADAM will quickly respond with 8. Similarly we can type PRINT3 ^ 2, hit RETURN, and ADAM will respond with 9 (two 3s are multiplied together).

You can also use negative exponents 2 ^ (−3) means 1 divided by 2 ^ 3. You can even use fractions or decimals as exponents, but we won't have much use for these advanced notions of exponents in this book.

Self-Check 2

Have ADAM compute the following numbers and print them on the screen.

1. 2^{16} (This is the amount of main RAM memory in ADAM.)

2. 2^{10} (In chapter 2, we called this number 1K.)

3. 3^{85} (This number is too big for ADAM. Reduce the exponent until ADAM gives an answer rather than an error message.)

4. 3^{-85} (This number is too small for ADAM to handle. Increase the exponent until you get an answer.)

5. A number is called a *perfect square* if it is equal to some number times itself. Another way of saying this is: A number is a perfect square if the number, for example 4, raised to the power .5 is an integer (a number with nothing after the decimal point). Which numbers between 80 and 120 are perfect squares?

If you worked through Self-Check 2, you may have noticed that you got some strange answers. For example, when we tried:

```
PRINT 3^85
```

ADAM responded with ILLEGAL QUANTITY ERROR. When we tried PRINT 3 ^ 80, ADAM gave us the response 1.47808825E+38. This is called *scientific notation* or *exponential notation,* and it isn't hard to use. It is very convenient for extremely large and extremely small numbers. Just think of E as meaning "move the decimal point to the right and add zeroes if necessary." But how many digits should you move the decimal point? It's exactly the number following the E.

Example 2.35E+7 means 23,500,000. We moved the numbers two places to the left and then added five zeroes (because E was 7 and we used two of the places to move the decimal point past the 3 and the 5).

Example The number 3^{80} the ADAM computed is really

1,478,088, 250,000,000,000,000,000,000,000,000,000,000.

Actually, ADAM only gives us the first nine digits of this number correctly. We think that writing out a few numbers like this shows the usefulness of exponential notation better than we could ever explain it. Imagine trying to count the digits to make sure that you have the right number of zeroes for 1.0E+100.

Example 1.056E00 means don't move the decimal point. It is the same number as 1.056.

ADAM uses the E notation for any numbers larger than 999,999,999. In fact ADAM usually just calculates the first nine digits accurately and then fills in the rest with zeroes. We say that ADAM computes to nine *significant figures.* In general, the number of significant figures is the number of digits that are actually computed before a computer starts replacing all the rest of the digits with zeroes. For example, if you ask ADAM to multiply 12223334445566 times 666677778889999 you get 8.14902546E+28. This number is not really the answer. The correct answer—which we refuse to compute because it has twenty-nine digits—has the last digit equal to four instead of the zero which ADAM gives us.

Don't worry too much about the fact that ADAM only is accurate to nine digits. Engineers say that if they could measure the circumference of the earth with nine digits of accuracy, they would know the circumference with a possible error of about an inch.

Exponential notation works the same way for things like 1.7E−5. Just move the decimal five places to the left this time so you end up with .000017.

Self-Check 3

A. Change the following numbers to E notation:

1. −30568
2. .00054
3. .10054
4. .00000000023

B. Change the following numbers in E notation to decimals:

1. 1.05E+2
2. −3.6207E−7
3. 2.1E−4
4. 8.604E+14

C. Ask ADAM to print the following:

1. 1.3E7
2. 5.6E−9
3. 1.234E4

D. Experiment by taking different numbers until you find just how large a number ADAM can accept.

If you want to do more complicated calculations than just using two numbers at a time, we suggest using parentheses. Otherwise you must follow the *hierarchy of operations,* described below. In our own work, we usually use parentheses. To us they are clearer and so we make fewer mistakes. In case you may have forgotten, parentheses order mathematical operations. Something like 3+(5*4) means 23, because first we do the operation within the parentheses (5 times 4) and only then add the 3. If we wrote (3+5)*4 this would give us 32, because first we add the 3 to the 5 to get 8 and only then multiply by 4.

When we use parentheses, the operations inside the parentheses are performed in the order just described, but the innermost terms are evaluated first.

Example ((((4*6)+4)*8) ^ 2)+9

```
=          4*6        =   24
     +      4        =    4
                        ----
                          28
     *      8        *    8
                        ----
     ^      2          224   ^ 2 = 50176
                                  +    9
                                  -------
                                  50185.
```

People who work with computers become experts at counting left and right parentheses. A missing parenthesis or one that is out of order totally confuses the computer. The best way to check your parentheses is to add one for each left parenthesis and subtract one for each right parenthesis. Start counting at the left and move right. If you never go below zero and end up at zero after counting all the parentheses, you've done fine.

Self-Check 4

A. Use ADAM to compute the following and print them to the screen:

1. (3*6) ^ 12
2. 3*(6 ^ 12)
3. 1 − (2 ^ 5)
4. (1−2) ^ .5
5. (3+4)*(1+2) ^ 6
6. ((3+4)*1)+(2 ^ 6)

In Self-Check 4, you may have noticed that ADAM was asked to compute the square root of −1. Since this is ordinarily not permissible in arithmetic, ADAM can't do it.

Despite our comments about trying to use parentheses whenever possible, it is still very useful to know the natural order in which all computers, including ADAM, perform arithmetic operations. The order is:

1. exponentiation

2. multiplication and division

3. addition and subtraction

We'll call these numbers "levels."

Operations of the same level are performed from left to right. Here are some examples:

2+3*2 = 8 Since multiplication is level 2, it is performed before the level 3 operation of addition.

2*3 ^ 3 = 54 Since exponentiation is at the highest level, we first calculate 3 ^ 3. This is (3)(3)(3)=27. Then we multiply the result by 2.

4/2*6 =12 Since multiplication and division are on the same level, we do the division before the multiplication.

Now let's look at a hard example. After this you will never have any trouble with the heirarchy of operations again (because after it you'll be convinced parentheses are easier).

3*2+6/8+4 ^ 2 ^ 2

First the exponentials are evaluated from left to right so 4 ^ 2 ^ 2 means 16 ^ 2 or (16*16) so we get: 256.

Next multiply 3*2 6.

then divide 6/8 .75

then add (lowest level) them together 262.75

Self-Check 5

A. Compute the following numbers by hand and on ADAM:

1. 4*6−8/9+3 ^ 2
2. 16/8/4/2
3. (((16+8)*4)/8+4) ^ (3*1)
4. 1+(−2 ^ .5)
5. 1+((−2) ^ .5) (What's wrong with this?)

This book is about programming and programming involves having the computer store certain commands and numbers and do various things with them. (*Store* is another buzzword. It means temporarily memorize.) If you have ever used a calculator that has a memory, you may have some familiarity with this. We want to show you how you can use a tiny portion of ADAM's memory in the same way that you used the memory on your calculator. If you haven't used a calculator, think about wanting to do a calculation of sales tax. Each time you compute the sales tax, you are multiplying by a single fixed number (for example in New York City, 8.25%). You don't want to constantly type the .0825, which is the decimal equivalent of 8.25%. You want to type it in once and call or send for it when you need it. Let's look at the following example.

Example Suppose we want to find out which of the following pizzas is cheaper.

	9 inch	**12 inch**	**15 inch**
Price	$6.00	$7.00	$8.00

You may not have thought of it exactly this way before but what you are doing is asking which of these pizzas has the least cost per square inch. So we need to calculate the area of various circular objects whose diameters are known. You may remember the formula from school. It involves using the number whose symbol is π, which is called pi and is pronounced

"pie." Its value is 3.14159265 (to 9 decimal places). It is multiplied by the square of the radius to get the area. In this case the pizzas are given by their diameters so we divide the diameter by 2 to obtain the radius. You probably have seen the formula:

$$\text{area} = \pi * r \hat{} 2.$$

To be able to do this calculation and so be able to compare costs we will need the number π more than once. We don't think it is much fun to type it in each time. What we want is to have ADAM remember the value for π and then call it when we need it—the way we send out for pizza when we are hungry. Remember that you must hit RETURN at the end of each line. This can be done even in calculator mode by first typing:

```
LET pi= 3.14159265
```

and then typing:

```
LET a = pi*(9/2)^2
```

What ADAM is doing here is very important. The first statement causes ADAM to set up a box in its memory whose name is *pi,* and in it placing the number 3.14159265. The second PRINT statement sets up a box called *a,* computes the area of the pizza and places that number in the box. This point is worth emphasizing.

> WHENEVER ADAM ENCOUNTERS A LET STATEMENT, THE SYMBOL ON THE LEFT SHOULD BE THOUGHT OF AS THE NAME OF A BOX AND THE DATA ON THE RIGHT SHOULD BE THOUGHT OF AS LIVING IN THE BOX. ONCE SOMETHING IS IN A BOX, YOU CAN ALWAYS SEND FOR IT BY USING THE NAME OF THE BOX THAT IT IS IN.

If we then type:

```
PRINT a/6
```

ADAM will print out the number of square inches of pizza that we buy per dollar when we buy a 9-inch pizza.

Self-Check 6

By changing the line LET a . . . , you can change the size of the pizza whose area you are computing. If you then retype the PRINT statement with the price of the new size of pizza, you can get the area of pizza per dollar for the other size pizzas.

A. Fill in the following table:

Size	9 inch	12 inch	15 inch
Square inches/$			

B. Which size pizza gives you the most pizza for your money?

BASIC uses the word *variable* where we have used the idea of the name of a box, but our analogy is a good one. If, whenever we use the word variable from this point on, you think, "that's the name of a box," you will never go too far wrong.

You can use any combination of letters and numbers for a variable as long as the first symbol ADAM encounters is a letter. However, ADAM only recognizes the first two characters and doesn't distinguish between upper- and lowercase letters. You have to be careful about using special characters as variable names—some special characters have special uses in variable names. Some variable names like LET and PRINT are forbidden because they are used as SmartBASIC commands. (*Character* is the

buzzword meaning a letter, numeral, or a special printable symbol like a punctuation mark or even a space.) For now, just use letters and numbers in variable names and don't worry about the forbidden names.

Example Try the following:

```
] LET Alfred = 10
] PRINT Alfred
] PRINT Al
] PRINT Alaska
] PRINT Alf
```

(ADAM should print 10 in response to all of these.)

To summarize, you can have ADAM do calculations by waiting for the SmartBASIC prompt] to appear, then typing PRINT together with whatever you want done. Press RETURN and the answer will appear. You can use parentheses if the calculations are complicated.

Next, if you need to use a number more than once you can define a variable (the name of a box) whose value (contents of that box) can be recalled whenever ADAM needs it. To recall what's in that box, you just use the name of the variable in your PRINT statement.

After you type in any line, you must hit the RETURN key to end the statement and tell the computer to act on it. In computerese, we say that we *enter* the line.

Typing HOME, followed by hitting the RETURN key, clears the screen and places the cursor at the upper left of the screen.

Summary

In this chapter, we showed you the following commands:

HOME	Clears the screen
LET	Opens up a box and fills the box with some value

PRINT	Displays something on the screen

as well as the symbols: + (addition), − (subtraction), * (multiplication), / (division), and ^ (exponentiation).

BUZZwords

CHARACTER	IMMEDIATE MODE
CURSOR	PROMPT
DIRECT MODE	SCIENTIFIC NOTATION
ENTER	SCROLLING
EXPONENTIAL NOTATION	SIGNIFICANT FIGURES
GIGO	STORE
HIERARCHY OF OPERATIONS	SYNTAX
	VARIABLE



Statements in BASIC

Introduction to Programming

In this chapter, we show you how to write a SmartBASIC program and use the printer to print the contents of the screen. You will also learn the following commands in SmartBASIC:

NEW	REM	INPUT
CONTROL/P	LIST	RUN
SAVE	LOAD	CATALOG
END		

as well as some of the uses ADAM has for the following punctuation:

" " quotation marks
; semicolon
: colon

Writing a program in BASIC is like putting ADAM on automatic pilot. In the last chapter we showed you how to use ADAM as a calculator—programming means having each statement processed automatically without having to fiddle each time. (*Processing* is one of the buzzwords in computerese. It means that ADAM carries out the instruction you gave it.) Let's look at a simple example.

Our First Program

First check to see that the SmartBASIC prompt] is on the screen. If it isn't, put the SmartBASIC tape into the data pack drive, press the COMPUTER RESET, and wait for the] to appear on the screen. Now type the word NEW and hit the RETURN key. Another] will appear. Type in the following program. Type it in exactly as we've written it. Be sure to leave spaces as shown and to hit RETURN after each line. Wait for the] to appear before typing in the new line. If you get any message from ADAM, you probably mistyped the line. Just retype the whole line including the number, and remember to hit RETURN at the end of the line. (Note: Don't hit RETURN after the word *tripled.*)

```
10 REM the first program
20 PRINT "Enter a number that you want doubled and tripled
when the ? appears."
30 INPUT a
40 PRINT 2*a, 3*a
50 END
```

When you have finished, another] will appear. Now type RUN and hit the RETURN. A short sentence explaining what to do when the **?** appears will print on the screen. Type in a number and hit return—ADAM will immediately print out twice that number, move the cursor (remember, that's the flashing line), and then type out three times whatever number you had typed in. You can try again with different numbers. Just type RUN

again and press RETURN. You won't have to wait very long for the ? to appear. Then you can proceed as before.

Since this is our first program, we will spend more time on it than usual. If you have some experience with BASIC, you might want to skim this whole chapter.

A BASIC program consists of many individual statements. Each one is like the statements we saw in the previous chapter. But, remember:

> IN A PROGRAM, EACH STATEMENT NEEDS A LINE NUMBER.

For reasons that we'll see shortly, it's a good idea to increase the numbers by 10. We want to go over what each line does in this program.

Line 10—This is called a remark or REM statement. This line is put in for the benefit of the programmer or anyone else who wants to read and understand the program. ADAM doesn't do anything with the contents of a line starting with rem. So a line like:

```
REM PRINT 2+2
```

will do nothing, even in calculator mode. Since the computer does absolutely nothing when it sees this line, REM statements are called *nonexecutable statements*. You might notice that in this paragraph we sometimes wrote rem in lowercase letters and in the program wrote in uppercase letters. We did this in order to point out that SmartBASIC does not distinguish between lowercase and uppercase letters. (In many versions of BASIC you must use uppercase letters.) There is no difference between typing:

```
INPUT A
input a
```

since ADAM interprets them as exactly the same statement. There are two exceptions to this rule. We'll see both of them a little later in this chapter.

It's easy to question the usefulness of REM statements. When we first learned programming it always seemed that our teachers used an enormous number of REM statements. Now that we have done more programming we realize their importance. The problem is that once a program is more than a few lines long, its easy to forget what it does or what you should do when strange question marks appear on the screen, which brings us to line 20.

Line 20—Up to this point, we've only used PRINT statements to do arithmetic. Here we are using a PRINT statement to type out a message before the **?** or prompt appears. (Remember, *enter* is the computerese word meaning type and then hit RETURN.) SmartBASIC will print anything exactly as it appears providing it is enclosed by quotation marks. This could be a word or a number. A good example to keep in mind is the following:

```
PRINT "2+2"
```

Try this. You should see the symbols 2 + 2. This is very different from what would have happened without the quotation marks.

ADAM displays only 31 characters per line on the television screen. In computerese, we say that these 31 characters form a *physical line.* There are twenty-four physical lines on your screen. A physical line is distinguished from a *logical line*, which is what you write next to a statement number. An example of this is:

```
200 PRINT "ANTIDISESTABLISHMENTARIAN"
```

This statement takes up more than one physical line on your screen though it is only one (logical) line in your program. In SmartBASIC, a logical line can be 129 characters long. This is a little more than four lines on your television.

AFTER TYPING A LOGICAL LINE IN A PROGRAM,
YOU MUST HIT THE **RETURN** KEY.

Line 30—This is an INPUT statement. When ADAM is running a program (another word for running that is sometimes used is *executing*) and encounters a line beginning with the word INPUT, it does two things:

1. It sets up a box in its memory whose name is whatever symbol(s) follows the word INPUT - in this case the box is called *a*.

2. It prints a question mark **?** on the screen.

Now you must type in a number and hit the RETURN key. By hitting the RETURN key, ADAM is told that it should take the number you entered and place that number in the box labelled *a*.

Line 40—This line tells ADAM to print the answers on the screen. The PRINT statement looks just like the PRINT statement in immediate mode except for the comma. Here we are asking the computer to print two different things. So we separate the things by a comma.

In a PRINT statement the first number is shown on the screen as far left as possible. The comma forces the second number to be shown on the same line but near the middle of the screen. If you ask that a third number be printed, it would appear at the beginning of the next line. (These printing locations are called *print zones*.)

Line 50—This is the END statement, which just tells ADAM to stop because the program is over. It is also a nonexecutable statement because it doesn't cause the computer to do anything. Although ADAM stops when it has no more statements to process, it is good programming practice to have an END statement.

To quickly summarize what we have discussed so far, a BASIC program consists of a number of statements, each one telling ADAM to do something (except for the nonexecutable statements). You begin programming ADAM by loading the SmartBASIC tape and pushing the COMPUTER

RESET on the top of the system unit. Then after a few seconds, a] will appear. Then type in the word NEW, hit RETURN, and you are ready to program. Enter the program and then enter the command RUN. If you haven't messed up, it does what you asked it to do. You can have only one program in ADAM's memory at any one time and entering the command NEW destroys whatever program may have been there.

Two features of SmartBASIC will make writing programs easier.

1. SmartBASIC makes no distinction between upper- and lowercase letters in commands and variable names.

2. SmartBASIC ignores spaces between words except, of course, if they occur inside quotation marks. The statements:

```
30 PRINT a     +     b
```

and:

```
30 PRINT a+b
```

are exactly the same to SmartBASIC. **Caution**: At least one space must appear after each command. (You can try PRINT3+4 to see what goes wrong.)

Learning to program is like learning anything else; you need to practice. In many ways programming is easier to learn than most other things. This may seem like a strange statement and you probably don't believe us. But remember this: a program consists of a bunch of simple steps, each one of which is very clear. To write a program that will solve your problem, first think about how you would solve it, then divide the solution up into simple steps before you start programming. Then translate each step into the appropriate SmartBASIC command.

This book has three purposes. First we will show you the commands that SmartBASIC can understand. Second, you will learn how to break up a problem so that it can be translated into those commands. Last, we will show you how to do the translation.

Translating the Solution to a Problem into a Program

Let's try learning this procedure by writing another short program. This one will add any two numbers that are entered. The steps are pretty easy:

1. Get the numbers in.
2. Add the numbers together.
3. Get the sum of the numbers out.

The BASIC program is:

```
10 REM this adds two numbers
20 PRINT "Please type in two numbers separated by a comma when
the ? appears.  Then hit return"
30 INPUT a,b
40 LET c = a+b
50 PRINT c
60 END
```

By the way, notice that since we are inputting two numbers, they have to be separated by a comma. (*Input* is computerese for placing something in ADAM's memory.) Notice as well that we used another variable, *c*, to add the numbers together. Strictly speaking we didn't need line 40. We could have just written PRINT a+b on line 40 and dispensed with line 50 altogether. We did it this way because we wanted to do an absolutely literal translation. There are almost always other possible ways to outline the program, and so there will be other SmartBASIC translations.

Most people are a lot more familiar with words then numbers. Let's try to pick up some of the ways ADAM handles words. First, we saw above that ADAM prints anything enclosed in quotations exactly as it appeared:

Example (immediate mode)

1. PRINT "I can't do that!"

   If we hit RETURN, the sentence, complete with exclamation point, is printed on the screen. We like to think that ADAM takes a picture of whatever is between the quotes so that everything is perfectly reproduced.

2. PRINT "2+2"

   (This won't get you 4)

3. We saw in our first program that the line:

```
20 PRINT "Enter two numbers . . .
```

   works the same way. ADAM prints out exactly what appears between the quotation marks.

Suppose you wanted to send out a lot of invitations to a party. The message is the same but you want to change the name of the person each time. This can be done by the following program (remember that you begin a new program by typing NEW and then hitting RETURN).

```
10 REM the first program with words
20 INPUT name$
25 HOME
30 PRINT "Dear "; name$
40 PRINT
50 PRINT
60 PRINT   "     I'm having a party tonight- Please come."
70 END
```

Try it. When the **?** appears, enter the name of the person you would like to invite. The name you entered will disappear and the screen will fill up with the invitation. Hold off for a second with the obvious question of how

to print it out. An invitation isn't much good if it only appears on your screen, but we want to single out the points that are worth picking up from this program.

Line 10—This is a general remark, as described for line 10 before.

Line 20—Notice the **$** (dollar sign) at the end of the word name. *name* is a perfectly good variable, but ADAM distinguishes in its memory between those boxes that hold numbers and those boxes that hold words. To ADAM, *name* and *name$* are very different. *name* is a perfectly good name for a box that will hold numbers (this kind of variable is called a *numeric variable*. *name$* (because of the **$**) can hold words.

Line 25—We saw the HOME command in the last chapter. It will erase whatever is on the screen. This puts us in a position to print our invitation. If you are wondering why it is line 25 rather than line 30, it's because we left it out at first and filled it in later. This is the main reason for increasing the line numbers by 10. It leaves room for changes or improvements. It also helps in correcting sloppy programming, which is why we used it.

Line 30—Notice the space within the quotes. This will be printed as a space on your screen.

> REMEMBER: ANYTHING WITHIN QUOTES IS PERFECTLY DUPLICATED.

Next notice the **;** (semicolon). This tells ADAM to print the next thing directly after what had come before. It won't move to the next print zone or space or carriage return before printing the next thing.

Line 40—A PRINT statement without anything following it acts as a carriage return, as does line 50.

Line 60—Since we wanted to indent, we put some spaces within the quotation marks.

Okay, now we have a program that prints an invitation on a screen. If we had a very long extension cord we could carry the television and the system unit and the printer to the friend's house and show them the invitation. This doesn't seem to be the best way to send an invitation, but that's all

we've shown you how to do so far. Of course, SmartBASIC gives us a way to print the invitation on the printer. If you hold down the gray key marked CONTROL and hit the P key, everything on the screen will be typed on the printer exactly the way it looks on the screen. We will usually indicate this combination of CONTROL and P by CONTROL/P. It doesn't matter whether the P is upper- or lowercase. This will print exactly what is on the screen. This would include any stray] or . that may be there. If a line is blank, CONTROL/P returns the carriage on the printer.

Self-Check 1

A. What does the following program do?

```
10 LET a$ = "HE"
20 LET b$ = "L"
30 LET c$ = "P"
40 PRINT a$,b$
50 PRINT a$;b$;c$
60 END
```

B. Run the program on page 62 again using the name of a different person.

C. Try a different form letter. For example, you might want to tell some companies that their computers made errors in computing your bill. If you ever plan to send this letter, use language that will not have you thrown in jail.

You can jazz up the invitation program by combining lines 40 and 50 in the following form:

```
40 PRINT:PRINT
50
```

Line 50 isn't needed any more. So, by typing in line 50 with nothing in it, we remove it. When SmartBASIC encounters a colon separating two statements on a single line, it treats each statement as though it had appeared on a separate line. For example, you will often see a line like:

```
2000 LET sal=20000: REM sal is salary
```

This is often done for the simple reason that when you introduce a variable it's a good idea to let other people reading your program know what the variable will stand for. Since the colon lets us combine two statements into a single line, we have done so here. An assignment (LET) has been combined with a remark (REM).

The use of the PRINT statement can be summarized as follows:

PRINT	with nothing else on the line acts as a carriage return.
PRINT a, b	prints the contents of the boxes *a* and *b*. The contents of *a* are printed at the far left of the screen and those of *b* is printed starting at the middle. The contents of these boxes are numbers.
PRINT a$,b$	is similar to the previous statement except that the boxes will now contain words.
PRINT a;b	prints the contents of the boxes *a* and *b* next to each other.
PRINT "xxxxxxx"	prints exactly the things between the quotation marks. This makes it hard to actually print a quotation mark. Most programmers use one or two apostrophes to get around this problem.
CONTROL/P	prints the contents of the screen on the printer.

Assignment Statements

An assignment (LET) statement is one you will use frequently. For this reason we think it worth spending a bit more time on. Some of its properties are somewhat surprising.

Look at the following statement:

```
LET i = i + 1
```

This statement makes no sense at all as a mathematical statement. We can subtract *i* from both sides and get 0 = 1! Of course, this is not a mathematical statement; it is a programming statement. To ADAM, it says there is a box (or place in ADAM's memory) which is labelled *i*. It contains some number (like 5). The statement tells ADAM to replace the number that is currently in the box with that number plus 1 (5 + 1 = 6). These statements do make sense and in some ways it is a shame that the equal sign is used for it. In other programming languages other signs are used, but people seem to get more annoyed than happy when they use them.

You should be aware that you can only do arithmetic on the right-hand side of a LET statement. In immediate mode you might try the statement:

```
LET i + 1 = i
```

just to see what you get.

```
LET i = a * b
```

is a valid statement. It will take the number in the *a* box, multiply it by the number in the *b* box, and place the product in the *c* box.

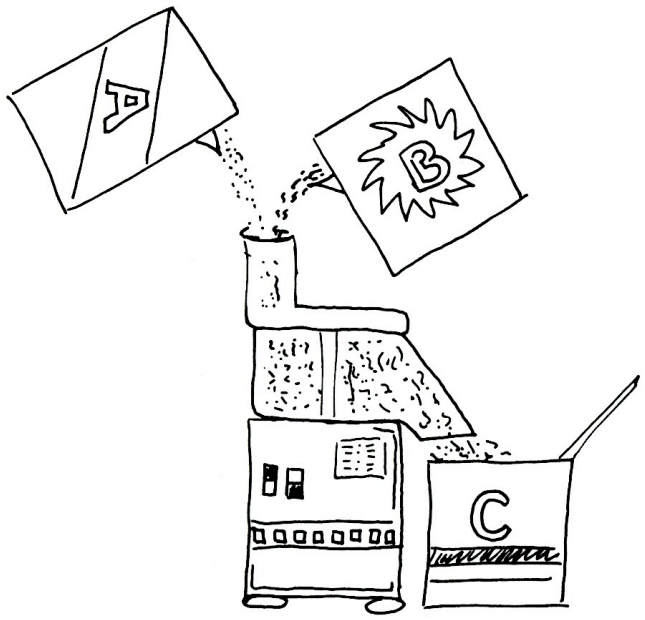


Fig. 4.1

As we saw in our second program, LET statements can be used to stuff boxes with letters or words, as long as the name of the box has a **$** following it. Anything in a box named using a dollar sign must be enclosed in quotation marks. For example:

```
LET a$ = "ADAM"
```

is quite valid. Strange things happen if either the dollar sign or one or both of the quotes are missing.

An assignment statement consists of two parts separated by an equal sign. The left-hand side is the variable name. Remember that this is the identifier for a box in ADAM's memory. The identifier can be over one hundred characters long, but only the first two count. Whatever ADAM encounters on the right-hand side of the equal sign is put in (assigned to) the box named on the left hand side.

When a variable is first encountered, it is automatically assigned the

value 0 unless you use an assignment statement to change the value. The statement:

```
10 PRINT adam
```

will print a zero on the screen. When SmartBASIC sees this command, it looks for a box named *adam* (or *ad*, because they're the same thing to SmartBASIC) and it doesn't find one. So it constructs a box named *adam* and puts a zero in it.

> ALL NUMERIC VARIABLES HAVE THE VALUE ZERO UNLESS ASSIGNED A DIFFERENT VALUE.

The zero acts as a placeholder. In computerese, we say that the *default value* of all numeric variables is zero.

Self-Check 2

In immediate mode, type in the following assignment statements one at a time. Hit RETURN. If SmartBASIC won't accept a statement, you should correct it and then enter either PRINT a$ or PRINT a, depending on which you have assigned.

```
1.  LET a$ = "ADAM"
2.  LET a = "ADAM"
3.  LET a$ = "ADAM
4.  LET a$ = "ADAM"
5.  LET a = ADAM
```

Tricks of the Trade (You'll see these sections occasionally. They'll give you tricks that programmers sometimes use to make their jobs easier.

These tricks are not a necessary part of programming. You can become an excellent ADAM programmer without using a single one.)

Some programmers get tired of using the word LET. For this reason, it is optional in SmartBASIC. For example you can enter *pi* = 3.1415965.

Alphanumeric Assignment Statements

Any variable that stores words can also store numbers or spaces. This kind of combination is called *alphanumeric*. The alpha stands for alphabet or letters and the numeric stands for numbers. The kind of variable with a dollar sign at the end is called a *string variable*. The alphanumeric characters that are stuffed into the boxes labelled with dollar signs are called *strings*.

String variables are very versatile. You can store almost anything in them. Of course you pay a penalty. Strings take up lots more room in the memory than numbers do. All of the following are legitimate SmartBASIC statements:

```
LET a$ = "Ronald Reagan"
LET b$ = "Queen Elizabeth"
LET c$ = "                "
LET d$ = "16 years old"
LET e$ = "123456789"
```

Since spaces and numbers are symbols, these can be printed and will appear exactly as shown with the quotation marks gone. Particularly interesting is the variable *c$* because it gives us a string of blank characters. These are often used as filler to position the display of some text.

Putting something new in a box (in computerese, this is called *changing an assignment*) destroys the previous contents of the box. This occasionally causes some problems. Suppose you want to switch the contents of the *a* box and the *b* box. A statement like:

```
LET a = b
```

doesn't work at all. What it does is duplicate the contents of the *b* box and place that duplicate in the *a* box. The previous contents of the *a* box are destroyed. So you can't do the switch. The following little piece of a program is worth learning. It shows you how to "swap" the contents of two boxes.

```
        .
        .
        .
        .
5000 LET c = a : REM save a temporarily as c
5010 LET a = b : REM copy b into a
5020 LET b = c : REM take a out of storage in c and put it in b
5030 REM now the contents of a have been swapped with the
         contents of b.

        .
        .
        .
        .
```

The variable *c* acts as a temporary storage location for the contents of the *a* box so that the information is not destroyed by line 5010. (We suggest making sure that you understand how it works before going on.)

More Sophisticated Programs and How to Save Them

We are now in a position to write some programs that are both more sophisticated and more interesting. Furthermore, we can display the results on the screen in a way that makes them easy to understand. (The buzzword for the way to set up a display on a computer is *format*, and we say that we have *formatted the output.*)

It may seem silly, but let's jazz up our simple program to add two numbers so that the display is nice.

```
10 REM this program gives a well-designed output
20 PRINT "type in two numbers separated by a comma and hit return"
30 INPUT a, b
40 PRINT a;" + " ; b; " = "; a+b
50 END
```

By now you should know that you must hit the RETURN key to enter a line after typing it. You should see that we have used semicolons in line 40 instead of commas. When we used commas in a PRINT statement, each item separated by a comma was printed in a new print zone. The semicolon forces the items to be printed right next to each other. If you look more carefully you'll see that we leave some spaces around the plus sign and the equal sign in line 40. If we didn't, ADAM wouldn't display any spaces either. If you want to see what would happen, retype line 40 without the spaces inside the quotation marks, hit RETURN, type RUN, and hit RETURN again. From now on, we won't keep telling you to hit RETURN at the end of each logical line; but remember nothing will happen until you do.

Here is another example:

```
10 REM this program computes the average of 4 numbers
20 PRINT "type in 4 numbers separated by commas, then hit return"
30 INPUT a,b,c,d
40 LET av = (a+b+c+d)/4
50 PRINT "your average is "; av
60 END
```

Again in this example, the space within the quotation marks, after the word *is*, is important.

In any program it's a good idea to tell people what is expected of them when the prompt **?** appears. Whenever we see a "bare" **?**, we get confused. We don't know what the programmer wanted us to do or what the program will do. Because a statement explaining the prompt is so important, SmartBASIC allows a simple way to explain what you want. The explanation is called an *input prompt*. To explain how it is used, we will show you how to eliminate lines 20 and 30 of the previous program. First enter:

```
25 INPUT "type in 4 numbers separated by
   commas, then hit return ";a,b,c,d
```

To get rid of line 20, enter in the number 20. Do the same for line 30. Now when you run the program you will see the statement from line 25 followed by the flashing cursor. No question mark appears on the screen when you use an input prompt. You can always put one inside the quotation marks or (as we did) leave a space before typing the second set of quotation marks.

By now you may have written a program you would have liked to save. Each time we enter NEW we destroy the program which is in ADAM's memory. The only way to save a program is to record it on the digital data pack. Then we can always play it back into the memory. When we play it back, we get the same result as reentering it, except we don't introduce any new typing errors. A few simple commands are used to record and playback. Here they are, with instructions on how to use them.

We'll first show you how to SAVE a program. If you haven't already removed the SmartBASIC data pack from the drive, remove it now. Insert the blank data pack that came with the machine. Now think of a name that has something to do with the program. For example, *average* seems like a good name for the program we just wrote. Since we have just finished running a program, a] should be on the screen and the cursor _ should be next to it.

Enter SAVE average (so the screen looks like this):

```
] SAVE average
```

Hit the RETURN key and wait a few seconds. You should be able to hear the data pack drive motor running and then stop; then the] will appear again. That's all you have to do to record or SAVE your program. If SAVE doesn't work, you'll see:

```
I/O ERROR
```

on the screen. I/O ERROR means that something has gone wrong with information, in this case a program, which is being moved into or out of memory. It stands for *input/output error*. Sometimes trying SAVE again works. If the data pack is not positioned correctly (for example, if it is in

kinds of errors. Sometimes SmartBASIC will suggest the proper form for a line. It does this, for example, when you forget to type a variable in an input statement. The errors that really cause problems are the statements that ADAM can understand but which don't do exactly what you expect.

The first step in fixing a program is to check that what is in ADAM's memory is what you wanted to be there. The command LIST will print on the screen whatever program is in ADAM's memory at the time. When ADAM lists a program, all commands appear in uppercase letters, all variable names are in lowercase letters, and the lines are printed in numerical order. This is how the program will be listed no matter how you originally entered it. You can look at line 20 alone by entering:

```
LIST 20
```

If you want to look at the lines from 20 to 50 you can enter:

```
LIST 20 - 50 or LIST 20,50
```

Now check out each line carefully. Is it what you wanted? Does it fit with the preceding lines?

The debugging trick we find most successful is to have a friend look at the program. Of course, the friend should know SmartBASIC. Someone else looking at a program often finds errors that we are blind to. The last resort is to trace the operation of a program by hand. We'll look at tracing programs and some other ways to debug them in Interlude 2.

Summary

In this chapter you started programming. We showed you how to use the following commands:

NEW	Erases an old program. This clears the way for you to begin writing another program.

INPUT	Sets up and names a memory location (box). Your response to the input request fills up this box.
RUN	Starts automatic processing of your program from the lowest numbered line.
END	Tells ADAM to stop processing the program.
CONTROL/P	Prints a "photograph" of the text appearing on the screen.
LIST	Prints out each line of your program in numbered order, not necessarily in the order you entered it.
SAVE	Saves your program on the digital data pack.
LOAD	Recalls a previously saved program from the digital data pack.
CATALOG	Tells you what is on a digital data pack.

We showed you how to use a colon to put more than one statement on a line and how to control the way results are printed using commas and semicolons.

BUZZwords

ALPHANUMERIC

ASSIGNMENT

BOMB

DEBUGGING

DEFAULT VALUE

ENTER

EXECUTING

FORMATTING

I/O ERROR

LOGICAL LINE

NONEXECUTABLE STATEMENT

NUMERIC VARIABLE

PHYSICAL LINE

PRINT ZONES	STATEMENT
PROCESSING	STRING
RUNNING	

INTERLUDE 

Editing Programs

Most people aren't letter-perfect typists. Unfortunately, when entering a program, nothing less will do. However, ADAM has many built-in commands that ease the pain of correcting errors. If you learn the techniques given in this brief interlude, then entering your programs correctly will be easier—or at least less frustrating. Of course what we will show you here isn't absolutely necessary. You could program ADAM by being extremely careful, and retyping lines as needed.

If a program is more than a few lines long, sketch it on paper. Some people advocate completely coding the program before they even sit down at the keyboard. (*Coding* is the buzzword for writing a program.) Although completely coding a program before entering it is a good idea, few people (including the authors) do it for all their programs. Usually a detailed outline is enough to get you started.

Once you've outlined your program and done all the necessary prep-

arations, you start typing. Unfortunately, the first thing you type might look like:

```
10 LRINT "HELLO"
```

(You might want to type this incorrect line—it will make it easier to follow what we say. If you do, don't hit RETURN.) The *L* has to be changed to a *P*. It is natural to hope there is a better way to correct this error than by retyping the whole line. If you've already hit RETURN, you will get an error message, but error messages are normal when you use a computer. They don't hurt ADAM and they don't hurt you. If you haven't yet hit RETURN then ADAM has an easy way to correct this type of error.

Before we can show you how to fix this simple error, it's a good idea to (re)familiarize yourself with the ARROW or CURSOR CONTROL keys on the right-hand side of the keyboard (see figure I.1). These keys are needed to use ADAM as a word processor. What they do is move the cursor (that's the flashing underline) around the screen. ↑ moves it up one line and ↓ moves it down one line. → moves the cursor one space to the right and ← moves it one space to the left. The BACKSPACE key has the same effect as ←. The HOME key moves the cursor to the top left-hand corner of the screen.

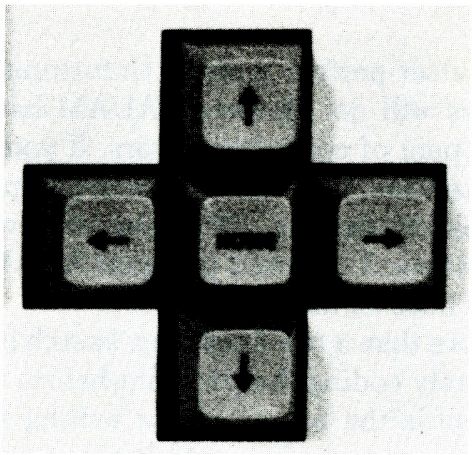


Fig. I.1 The cursor keys on the keyboard.

Now you can correct that mistyped line. Hold ⇦ down until the cursor lies under the *L*. Press the *P*. We say that you have *overwritten* the *L* with the *P*. The screen should look like this:

```
10 PRINT "HELLO"
```

The underline shows where the cursor is. Now hit RETURN. As with many illegal lines, ADAM will point out your error after retyping the line it received.

> WHEN YOU HIT **RETURN**, EVERYTHING THE CURSOR HAS PASSED OVER IN MOVING TO THE LEFT HAS BEEN FORGOTTEN. THIS MEANS THAT EVERYTHING TO THE LEFT OF THE CURSOR HAS BEEN DELETED.

You will have to retype the whole line. There is a way out. To see it in action retype the whole line with the same error. Now just as before overwrite the *L* with the *P* but *don't hit* RETURN *with the cursor in the middle of the line.*

Instead, use the ⇨ to move the cursor back until it passes over all the letters in the line. This is easy to do because any key repeats if it is held down. Hold down ⇨ until the cursor passes the last character on the line. Now hitting RETURN sends the correct line to ADAM's memory. By typing LIST 10 and hitting RETURN, you can check that the line has been "memorized" correctly.

Holding down a key was helpful in moving the cursor, but sometimes holding a key down too long causes problems like:

```
10 PRINTTT "HELLO"_
```

(The underline shows where the cursor might be when you discover the error.) This is an easy error to correct because SmartBASIC doesn't care about spaces except inside quotation marks. Just use ⇦ to move the cursor

below the second *T*. Press the space bar twice. Now the screen looks like this:

```
10 PRINT  _"HELLO"
```

Again the underline shows you where the cursor is. Just as before, hold down ⊡ until the cursor has passed over the remaining characters and then hit RETURN. The correct line is now in ADAM's memory. When you enter LIST 10 you will see that ADAM has even removed the extra spaces.

The next two methods for editing lines are a bit unusual—as far as we know, only ADAM has them. They only affect a *physical line*, a line on your screen, as opposed to a logical line. We stress this because it is easy to forget and, if you do, the methods we're going to show you will make things worse instead of better.

Suppose you typed:

```
10 PRINLT "HELLO I'M ADAM THE
FRIENDLY COMPUTER"
```

This takes up more than one line on your screen. You want to remove the extra *L*. Again using ⊡, move the cursor back (notice it will move up a line) until it is under the *L*. Hold down the CONTROL key and then press the letter *O*. (We'll call this CONTROL/O.) The extra *L* will disappear and the cursor will stay where it was. Now use ⊡ to move the cursor past all the characters and hit RETURN. If you type RUN, you'll see that there is an extra space between the words THE and FRIENDLY. When the *L* was removed, all characters to its right moved one space to the left. This left a space at the edge of the screen. Normally a space doesn't matter, but it does here because the space was between quotation marks.

> REMEMBER: **CONTROL/O** ACTS ONLY ON A PHYSICAL (OR SCREEN) LINE. IT DELETES THE CHARACTER ABOVE THE CURSOR AND LEAVES A SPACE AT THE END OF THE SCREEN LINE.

Often you want to insert one or more spaces into a line. This can be done and works much like CONTROL/O, but now whatever was at the end of the line disappears. Suppose you typed:

```
10 PRNT "I'M ADAM THE FRIENDLY
COMPUTER"
```

and you want to correct this by inserting an *I* in the right place. As before, use the ⇦ to move the cursor below the *N*. Hold down the CONTROL key and hit the *N* (this is called CONTROL/N). A space suddenly appears between the *R* and the *N*, but the cursor doesn't move. (If you pressed twice, two spaces would appear.) Now you can fill in the *I* and move the cursor to the end of the line. You might think everything is fine, but it isn't. The *R* has disappeared from the first line. If you want to put it back, move the cursor until it is at the first position of the second line. Use CONTROL/N to insert another space before the space that is there already. Now type in the *R* that got lost. Go past all the characters and then hit RETURN. The line is now correctly entered in ADAM's memory.

> REMEMBER: CONTROL/N INSERTS SPACE ONLY ON A PHYSICAL LINE AND ANYTHING ON THE END OF THAT LINE IS LOST. YOU MUST REINSERT THE LOST CHARACTERS ON SUBSEQUENT LINES.

Trick of the Trade Suppose you have a line that is garbled beyond repair and you don't want to waste time getting the inevitable error message. You can make ADAM forget what you typed by holding the CONTROL key and pressing the X. CONTROL/X prints a backslash (\) and the cursor moves to the beginning of the next screen line. You can now retype the line.

Another use of the CONTROL key (a complete list is given in Appendix A) is CONTROL/L. CONTROL/L acts like the HOME command (not the HOME key!) and clears the screen. Unlike the HOME command, it works equally well in the middle of a line. Be careful using CONTROL/L because it also removes whatever you had previously typed from the screen. If you use

CONTROL/L when you are in the middle of a line, ADAM assumes that whatever you now type should follow the lines that weren't erased by CONTROL/L. This leads to garbled lines and an error message. Play it safe and hit RETURN before CONTROL/L.

The arrow keys can also be used to fix or change lines of a program that have already been entered in ADAM's memory. Before we can show you how to do this, it's best to learn more about the LIST command discussed in chapter 4. Suppose you have written a long program which doesn't run. This is a common problem, so don't panic. You think the problem lies somewhere between lines 100 and 140. If you just use the command LIST the whole program will be printed on the screen and it is hard to isolate individual pieces. If the program is too long to fit on the screen, the lines you want to see may just scroll (or run) off the top of the screen. Table I.1 shows ways to use the LIST command to display any portion of your program.

LIST	Displays all or part of the program currently in memory.
Format	
LIST	lists the entire program starting from the lowest numbered line.
LIST *line*1, *line*2 or LIST *line*1 - *line*2	will list the lines between between line 1 and line 2.
LIST *line*, or LIST *line*-	will list the lines starting at the specified line and ending at the last numbered line.
LIST, *line* or LIST -*line*	will list lines starting at the first numbered line and ending at the specified line.
Examples	
LIST 100	will list only line 100.

LIST -100	will list all lines of the current program with line numbers up to and including 100.

VARIANTS OF THE LIST COMMAND
TABLE I.1

The format of table I.1 is a general format for tables showing you how commands can be used. These tables are extremely handy. We needed one when we wrote this book, so we made one up. Our *Reference Guide* for the ADAM is included in this book. We think it is a useful summary of all the commands in SmartBASIC.

By checking the table you find that you can list lines 100 to 140 by typing LIST 100-140. The display might look like the following (don't worry about what it means):

```
100 REM this part of the program prints
    headings
110 PRINT "HOURS";"SALARY","TAXES"
120 PRINT:PRINT:PRINT
130 PRINT H, 6*H, .7*H
140 GOTO 1000
```

When we ran the program, HOURSSALARY was displayed on the screen and this gave us an idea of where to look for an error. Looking carefully at this fragment, we saw that the semicolon on line 110 should be a comma. (SALARY should appear in the print zone after HOURS.) It is easy to fix this. We used the up and down arrows to position the cursor under the first digit of the number 110, the first *1*. Using the same methods as before, we moved the cursor to the right until it was under the semicolon and overwrote the semicolon with a comma. We finished exactly as before (by holding down ⊡). We had to be careful to release the arrow and hit RETURN before the cursor got to the next numbered line. Otherwise the extra characters would have been added to line 110. This would have made a mess of line 110 and led to an error message.

To summarize these methods of correcting blocks of lines: everything that is on your screen can be changed (in computerese, the expression is *the screen is live*). By using the arrow keys, CONTROL combinations, and overwriting, whatever was on the screen or anything you have listed can be changed. This allows you to make changes that would otherwise be tedious.

Trick of the Trade: There is one other way to correct errors. Although it is the most powerful method of all, it has the disadvantage that you can't see the changes as they happen (as you could with CONTROL/N or CONTROL/O). Keeping the following analogy in mind helps: when you use ◄ to make the cursor move left over a character, ADAM "forgets" the character; and as you use the ► to make the cursor move right over the character, ADAM "remembers" it.

Suppose you typed:

```
10 PRINT "I'M ADAMMM THE FRIEN
   DLLLY COMPUTER"
```

and you haven't hit RETURN. You could correct this by repeated (and careful) use of CONTROL/O. However, there is an easier way. When you hold down the CONTROL key and press the ► (we'll call this a CONTROL/►) this will still move the cursor *but ADAM will not "remember" the characters that the cursor moves over.* So hold down the ◄ until the cursor is under the second *M*. The screen would look like this:

```
10 PRINT "I'M ADAMMM THE FRIEN
   DLLLY COMPUTER"
```

While holding the CONTROL key, tap ► twice until the cursor is at the space (between ADAMMM and FRIEN). Because you used the CONTROL/► , ADAM does not remember the two *M*s. To eliminate the extra *L*s do the same thing. Using only the ► , move the cursor until it's below the second *L* and hold the CONTROL key while tapping the ► twice until the cursor is under the *Y*, release the CONTROL key, and use the ► until the cur-

sor has moved past the ". Hit RETURN and the correct line is recorded in ADAM's memory, although it certainly isn't on the screen. You can use the command LIST 10 to check whether the correction procedure worked. The CONTROL/ ◄ combination works the same way, only now ADAM remembers rather than forgets the characters that the cursor moves over.

This leads to a (very tricky) way to use CONTROL/◄. Suppose you had as line 10 of your program:

```
10 PRINT "HELLO I'M A COMPUTER
   "
```

and you wanted to change this by adding the word FRIENDLY. Enter LIST 10 so the line is displayed. It should look like this:

```
10 PRINT "HELLO I'M A COMPUT
   ER"
```

Using the ► start at the beginning of the line and move the cursor until it's below the *C*. Now tap the ▲. The cursor has moved up one screen line. Now type in the word FRIENDLY so the screen looks like this:

```
                          FRIEND
LY 10 PRINT "HELLO I'M A COMPUT
ER"
```

The cursor is right in front of the *1*. Now hold the CONTROL key and tap ◄ until the cursor moves past the F in FRIENDLY. Notice how the cursor has moved up so it is again on the line above (with the FRIEND). Now release the CONTROL key and press the ▼ so the cursor is under the space. Use only the ► to move the cursor past the quotes. Although it's impossible to know until you enter LIST 10 again, the new corrected line is in ADAM's memory. Check it and see.

(The use of the CONTROL/arrow combinations are not necessary to cor-



Changing the Order of a Program

In this chapter you will learn how to use the following commands in SmartBASIC:

GOTO	IF-THEN
CONT	ON__ GOTO__

We will also explain some more uses for the colon (:) as well as how SmartBASIC uses the words AND, OR, and NOT. We will show you how to stop a program by using CONTROL/C and how to continue it.

The new symbols you will learn are:

= (equal)	< > (not equal)
< (less than)	< = (less than or equal)
> (greater than)	> = (greater than or equal)

(Always) Changing the Order

SmartBASIC processes each statement in its numerical order, line 10 before 20, etc., even if we typed them in the wrong order. But if programs only worked this way, we couldn't do much with them. A computer's power comes not only from its incredible speed, but also from its ability to change the order in which it processes the statements of your program. ADAM can skip a statement because of one thing or go back to an old line because something happens. Imagine a game-playing program that does the same thing regardless of what we do—such a game would be silly and the time spent programming it would be a waste. In this chapter we will learn most of the ways SmartBASIC lets us change the order a program runs.

Using GOTO

The first thing we can do is change the order directly. Instead of going from line 40 to line 50, suppose ADAM encounters on line 40 the statement:

```
40 GOTO 100
```

Think of the game Monopoly: ADAM goes directly to line 100 without passing (processing) any of the lines in between. A GOTO command is the first example of what is called *branching*. Think of a branch as a place in a program that's like a fork in the road, except it's the computer that decides which way to go. A GOTO is an *unconditional branch* and this buzzword means what it says—no conditions, just change. An unconditional branch is a detour in a road. There are other ways that we can change the order in which our program runs. For example, computers excel at doing the same thing repeatedly. Think of it as a turnstile revolving—each time the wheel

turns, ADAM does the same thing. Even in a small, inexpensive computer like ADAM, the internal clock can spin very fast. Moreover, ADAM can keep count of the number of times the wheel has turned and, by analogy, the number of times it has done any particular operation.

The simplest way this can happen is when line 20 sends us to line 30 which sends us to line 20 again, and so on. We can see this happening in the following program. Think about it but *don't* try to run it yet.

```
10 REM THIS PROGRAM DEMONSTRATES THE GOTO
20 PRINT "I will not be late"
30 GOTO 20
```

What does this do? Line 20 prints the sentence:

```
I will not be late
```

on the screen and then ADAM processes line 30 which sends it right back to 20, so another "I will not be late" appears. *This will go on forever.*



Fig. 5.1

Remember the old fashioned punishment of having to write "I will not be late" five-hundred times? Suppose our teacher believed in this sort of thing, yet was open enough to new ideas to let us use ADAM. A program such as the one we just wrote would quickly print enough to satisfy the cruelest teacher. If he or she wanted *exactly* five-hundred copies, either handwritten or computer-generated, then this program wouldn't work. We will soon see that ADAM can keep count, but for now let's talk more about this circular process. Since ADAM is going around in circles, the buzzword is *looping*. Looping just means doing something repeatedly by circling back in your program.

Let's get back to the program on page 91. We hope you didn't try to run it. Because if you did, you quickly filled up your screen with that silly sentence. If you haven't yet, try it now. Obviously you want to stop it. How do you stop a program that doesn't want to stop?

TO STOP ADAM IN THE MIDST OF AN OPERATION, HOLD DOWN THE KEY ON THE LEFT MARKED CONTROL AND HIT THE C KEY. THIS IS CALLED CONTROL/C. IF A ? WAS ON THE SCREEN, WHILE HOLDING DOWN THE CONTROL KEY PRESS THE C KEY AND THEN HIT RETURN.

(If you are wondering why hitting RETURN was necessary in the second situation, it's because ADAM was waiting for input. That's what the question mark means. So you have to hit RETURN to tell ADAM to accept input.)

On rare occasions, CONTROL/C will not stop a program. Don't panic, just put the SmartBASIC tape back in the drive and hit RESET. The program in memory will have to be reentered.

After ADAM receives the CONTROL/C message, it will first type BREAK IN LINE - -, Then a] will appear. When the] appears, you can:

1. Start the program from the beginning by typing RUN and (of course) hitting the RETURN.

2. Start the program from the point where it was interrupted by typing CONT and hitting RETURN.

3. Do something like LIST and try to find out what's wrong with your program.

4. Just consider the]as an excuse to do any legitimate thing with ADAM. For example, you could write a new program.

Our punishment program is not in any way damaging ADAM, it's just that you won't be able to do anything else until you stop this program. ADAM is "trapped in an infinite (or endless) loop." *Infinite loops* and *endless loops* mean that the computer keeps circling and doing the same thing forever. These words terrorize people who program large computers. Infinite loops waste time and, on large computers, time costs lots of money. We have said it before, but it bears repeating:

> YOU CAN'T HURT ADAM BY ENTERING ANYTHING FROM THE KEYBOARD.

Hitting CONTROL/C breaks the infinite loop in this program.

Back to the punishment program. If nothing else, we can try to satisfy the teacher by keeping a careful count and hitting the CONTROL/C combination exactly when the five hundredth "I will not be late" has appeared on the screen. Most likely, you won't be able to keep an exact count. It's a lot easier to have ADAM do it for you. ADAM can easily keep count. We just need one or two more commands. We'll look at those commands later in this chapter and then we'll finish off the punishment program with an exact count at the end of the chapter.

Another use of the GOTO (or unconditional branch) is to make a program repeat without having to type RUN each time. For example, the program from the last chapter that added two numbers can be made to repeat by adding one line. Try the following:

```
10 REM this program adds two numbers together
20 PRINT "When the ? appears, please type in two numbers
separated by a comma and hit return"
30 INPUT a,b
40 PRINT a+b
50 GOTO 20
60 END
```

This works very well but you must use the CONTROL/C combination to stop it. We will soon see a much better way to force it to stop. GOTOs are very powerful; it's a good idea to explain them when you use them. This helps other people who may read your program, and makes it easier for you to debug it. In the program we just wrote, a better way to write line 50 is:

```
50 GOTO 20:  REM this runs the program again
             by going back to start
```

(Here we are using the alternative way of putting comments into a program. Recall you can either use a REM on a different line, or, better still, you can put it where it is needed by using a colon.)

Self-Check 1

A. Which of the following statements are legal in SmartBASIC? Which are legal but silly?

```
 1.  GOTO 50
 2.  10 GOTO LINE #50
 3.  10 GO TO 50
 4.  500 GOTO #60
 5.  500,000 GOTO a$
 6.  30 LET a = 5 AND GOTO 45
 7.  33 GOTO 33
 8.  10 GOTO a
 9.  25 GOTO florida
10. 16 GOTO "FLORIDA"
11. 10 GOTO 15:  GOTO 20
12. 10 GOTO 11
```

B. What's wrong with the following program?

```
10 PRINT "HELLO"
20 GOTO 20
```

C. What will the following program do?

```
10 REM you should fill in this statement
20 LET s=0
30 INPUT "type in a number and hit return";x
40 LET s =s+ x
50 PRINT s
60 GOTO 30
```

D. The program given below is a maze of GOTOs. Without running the program, can you tell whether it ever reaches the END statement? (This is one of the reasons that some people don't like GOTOs. They call programs with GOTO statements "spaghetti programs" because they can wind around themselves.)

```
10 REM this program sends you wandering
   through the lines
20 GOTO 90
30 GOTO 40
40 GOTO 100
50 GOTO 10
60 GOTO 70
70 GOTO 30
```

(continued)

```
80 GOTO 20
90 GOTO 50
100 END
```

You can construct your own mazes just like this one. Try a few.

E. Run the following program:

```
10 REM a screen walk
20 PRINT "HELLO ";
30 GOTO 20
40 END
```

You can see that the screen fills up and the words seem to walk across the screen. Try the same program, but print a word or sentence with a different number of letters. Remember to leave the one or more spaces. Try this several times. Can you figure out when the words will walk and when they will stand still? Can you figure out why?

F. Write a program that will fill the screen with dollar signs.

G. Write a program that will print the numbers from 1 till you get tired and stop it using CONTROL/C.

You Decide When to Change the Order

If you played the Buck Rogers game that came with the ADAM you have encountered another way to use the GOTO. Remember how the Buck Rogers game presented you with a menu from which you selected your skill

level and the number of players? (If you haven't played the game yet, now you have a good excuse.) The idea of giving someone a choice of options is called a *menu-driven program.* The person running the program can actually choose how it will run. Menu-driving is an easy way to make your programs *user-friendly,* meaning that a program doesn't need an expert to run it.

One of the easiest ways of making a program menu-driven in SmartBASIC is by the command:

```
ON____GOTO (some line in your program,
some other line, etc.)
```

You fill in the ____ with a variable name. The value of this variable tells you which line to go to. If the variable has value 1, you go to the first line in the list, 2 sends you to the second, etc. Menus are just one of the uses of the ON-GOTO command. First let's try a simple example.

```
10 REM the ON-GOTO command
20 INPUT "type in a number up to 5 and hit return ";a
30 ON a GOTO 80,100,120,140,160
40 PRINT "You didn't type in the right thing."
50 PRINT "     The program won't run."
60 PRINT " You can try again by typing RUN"
70 GOTO 170
80 PRINT "You chose the number 1"
90 GOTO 170
100 PRINT "You chose the number 2"
110 GOTO 170
120 PRINT "You chose the number 3"
130 GOTO 170
140 PRINT "You chose the number 4"
150 GOTO 170
160 PRINT "You chose the number 5"
170 END
```

This program lets you choose the particular PRINT statement to be executed depending on what you input. This is the first time that you can choose which statements are to be executed by interacting with ADAM. Line 30 makes the choice depending on your response to the input request in line 20. If you type in something that the program cannot accept, line 30

has no effect and lines 40 to 70 are executed. They will tell you that your choice isn't allowed. These lines trap your mistake and the message printed on the screen tells you what will happen. *Trap* is, of course, another of our buzzwords. You set up a place to go if a program receives something it can't handle.

This program really doesn't do much, except demonstrate the ON-GOTO command. Let's write a more interesting menu-driven program. It will convert either Fahrenheit to Celsius or vice versa, depending on whether the person types in a 1 or a 2.

The program must have four steps:

1. Display the menu so that a choice can be made.

2. If the number 1 is typed, do a Fahrenheit to Celsius conversion.

3. If the number 2 is typed, do the opposite conversion.

4. Display the results in a meaningful way.

Finally we need to recall the formulas for the conversion:

Celsius to Fahrenheit $F = (9/5)*C+32$

Fahrenheit to Celsius $C = (5/9)*(F-32)$

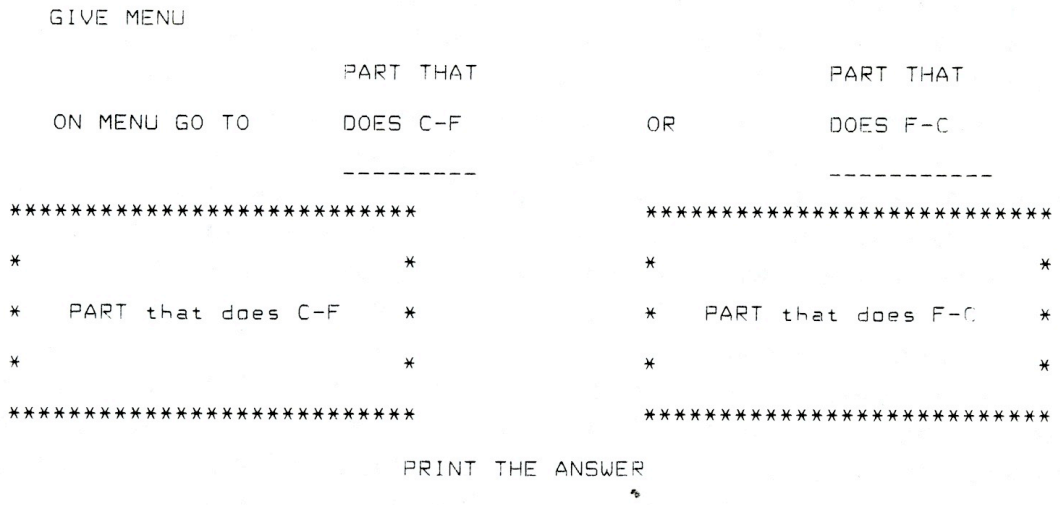
Here is the program:

```
10 REM an example of menu driving
20 HOME
30 PRINT "This program will convert between the two major ways
of measuring temperature"
40 PRINT " 1.  Fahrenheit to Celsius"
50 PRINT " 2.  Celsius to Fahrenheit"
60 PRINT "Type in the number of the conversion you want and
hit return"
70 INPUT m: REM M is the number chosen from the menu
80 ON m GOTO 1000,2000
90 PRINT "You typed in something strange. The program will
end."
100 PRINT "Enter run if you want to try again."
110 END
1000 INPUT "Enter the temperature in degrees Fahrenheit ";f
1010 LET c = (5/9)*(f-32)
```

(continued)

```
1020 GOTO 3000: REM 3000 is the location of the print statement
2000 INPUT "Enter the temperature in degrees Celsius "; c
2020 LET f = (9/5)*(c)+32
3000 PRINT c; " degrees Celsius is the same as ";f;" degrees
Fahrenheit"
3010 GOTO 110: REM every program should end by going to end
```

A program like this looks extremely weird. The END statement is in the middle, and the last statement tells you to go to the END statement. That's okay except for the location of END. The program line numbers look like they were chosen at random, rather than the nice pattern—10, 20, 30, etc.—of all previous programs. You might reasonably wonder whether the authors, the editors, and the publisher have lost their minds. Not so. Try to imagine how we (or you) would write this kind of program. What we did was take the English description and, on a piece of paper, write something like:



Then we wrote each part. Only toward the end did we know which line numbers should be assigned to each part; then we filled in the line numbers in the ON-GOTO command.

Think about it this way: when we have something hard to do we first try to divide it into several smaller jobs. Each one will (hopefully) be much simpler than the original task. Usually they have to be done in some reasonable order—you don't put a roof on a house before you build the foundation. Writing a complicated program is done the same way. Each piece

shouldn't be complicated—just long enough to do one thing. Then you figure out how to tie the separate parts together with a part that acts as a referee. This part should contain the END statement, so it's not surprising that in our program the END occurred in the middle.

Enough talking in the abstract about how to write programs. Let's see how this one works . . . Lines 10 through 110 control the flow. They give the directions for the other parts, and so form the main controlling piece. The first six of these lines (lines 10 to 60) just do housekeeping. They tell us what the program is about, clear the screen, and print the menu. In line 70, we input the menu number. This is the code number for the operation we want the program to do.

Line 80 interprets the menu number and sends the action off to the piece that does the conversion. Lines 90 and 100 are just a trap in case you type in something that can't be understood by the program. At least line 110 is a statement we've seen before. If we want a Fahrenheit to Celsius conversion we type in a 1 and line 80 sends us to the part formed by lines 1000, 1010, and 1020. These lines do the conversion and send us on to another part to print the results. The part that prints has two lines, numbers 3000 and 3010. The first prints the result and the second sends us to the end. If we had chosen the other conversion we would have taken a different path through the pieces. Each of the pieces performs a different task, and the part that orchestrates the whole thing precedes all others.

It is usually a good idea to use completely different line numbers for each different part—we used lines 10 to 999 to do the main controlling and 1000 to 1999, 2000 to 2999, and 3000 to 3999 for the other parts. Of course, we didn't use all these line numbers, but this way we can change lines in each part without wrecking the whole program. Many people write the parts separately and test them for errors before putting them together. We will explain how to test programs in Interlude 2. In large programs this kind of testing is essential; in medium-size ones, it usually helps.

By the way, the buzzword for part is *module* and so what we have described above is called *modular programming.* Our Fahrenheit to Celsius conversion is the first example of the method of modular programming and, for this reason, professional programmers would say it's the best-written program you have so far seen.

We said a lot about the way this program was constructed. It's worth summarizing.

1. A modular program is a program written in blocks.

2. You write an outline (in English) so that each step in the outline corresponds to a block (piece, module).

3. You try to do a word-for-word translation of an outline into a program.

The outline gives a method for solving a problem; the program gives that solution in SmartBASIC statements corresponding to the parts of the outline.

Modular programming is most easily done using subroutines. We won't be introducing them until chapter 12; but whenever possible we will write programs in modular form, so that you get used to seeing it.

Self-Check 2

Programs

1. Write a menu-driven program that will change pounds to grams and vice versa. (There are about 454 grams to a pound.)

2. Do the same thing for kilometers to miles. (There are about 1.61 kilometers to the mile.)

3. Do the same thing for liters to quarts. (There are about 1.056 quarts to the liter.)

4. Tie parts 1, 2, and 3 together in one menu-driven program.

Changing the Order (Sometimes)

GOTOs work blindly—they change the order regardless of what the program may have encountered or done up to that point. Often we want to reorder the processing of statements depending on what the program has

already done. In our everyday lives we do the same thing. At a baseball game if the umpire makes one kind of call we shout something impolite; otherwise we wait for the next action. ADAM uses a command called an IF-THEN to do something much like this.

Let's look at a few examples of the use of IF-THEN.

```
10 REM THIS PROGRAM DEMONSTRATES THE IF-THEN
20 LET a$ = "ADAM"
30 IF a$ = "ADAM" THEN PRINT "that's the name of an APPLE
   eater"
40 END
```

ADAM uses line 30 to check whether the contents of the box named *a$* in its memory contains the word ADAM. Since *a$* indeed contains the word ADAM (because of line 20), it immediately obeys the command following the word THEN. It prints:

```
that's the name of an APPLE eater
```

ADAM checks the first clause (called, naturally, the *if-clause*) and, whenever that clause is true, ADAM does whatever follows. What happens when it's not true? ADAM then skips to the next numbered line and does whatever it finds there. For example, try the following program:

```
10 REM this program demonstrates what happens when we
don't have a true "IF-CLAUSE"
20 LET a$= "CLARK KENT"
30 IF a$= "SUPERMAN" THEN PRINT "He can fly"
40 END
```

What happens? If you try it, you see that *nothing* happens. The box *a$* did not contain the word SUPERMAN, so the *then-clause* was never executed. (You may know that Clark Kent's real name is Superman, but remember, ADAM knows exactly what it is told—no more and no less.)

IF____1____ THEN____2____

An IF-THEN has two parts. Part 1, which follows the IF, is called the *IF-clause.* ADAM tests the *IF-clause* to see whether it is true. If it is true, the statement following the THEN, which is part 2 and is called the *THEN-clause*, is executed. The *THEN-clause* can be any legal SmartBASIC statement. When the *IF-clause* isn't true, ADAM skips to the next numbered line and processes it.

There are certain shorthand forms of the IF-THEN statement. A statement of the form IF_____ THEN GOTO _____ may be shortened to IF _____ GOTO _____ or IF _____ THEN *line #*.

Often you want to test conditions other than equality. You do this using the relational operators listed below.

RELATIONAL OPERATORS

= means equal	< > means not equal
< means less than	< = means less than or equal
> means greater than	> = means greater than or equal

It's frequently a good idea to build into your program a few lines that check whether the operator—the person using the program—wants to run (or continue running) a program. In the following program fragment, the operator is asked to enter in the word "yes" if he or she wants the program to run.

```
10  REM This program fragment demonstrates a good way to
    begin a program

20  PRINT "When the ? appears, enter yes if you want the
    program to run.  Otherwise enter anything else to stop"
```

(continued)

```
30  INPUT s$  : REM S$ is our status check
40  IF s$ < > "yes" THEN GOTO 1000
  .
  .
  .
  .
  .
1000 end
```

What this program does is jump to the end if anything other than the word *yes* is entered. This idea will occur constantly; it's another way to make your program user-friendly.

Our shorthand allows us to use other forms of line 40. For example:

```
40 IF s$<>"yes" GOTO 1000
```

or:

```
40 if s$ <>"yes" THEN 1000
```

In either case ADAM will jump to line 1000 and end the program if you enter anything but the word *yes.*

More often than not, we use the relational operators to compare numbers that are the values of variables. Remember this means numbers that are sitting in different boxes in ADAM's memory. Always think:

variable	=	the name of a box
value of a variable	=	what's in the box

We can have ADAM compare the contents of one box with the contents of another. Try the following program:

```
10 LET a = 10
20 LET b = 0
30 IF a > b THEN PRINT  "a was positive"
40 END
```

This makes ADAM print "a was positive" on the screen. (How would you have ADAM print the sentence, "10 is positive"?) Now try the following:

```
10 LET a = 10
20 LET b = 0
30 IF a < b THEN PRINT "no negative numbers allowed"
40 END
```

This time nothing happens after you enter RUN. Why? (The reason the second program doesn't do anything is because 10 isn't less than 0.)

You can also use the relational operators for strings (words). Here they just mean coming before or after in dictionary or alphabetical order with all capital letters coming before all lowercase letters. A comes before B, A comes before AA, AA comes before AB, Z comes before a, etc. If we want to write a program to check if two words are in alphabetical order, the steps would be pretty clear:

1. Read the two words in.

2. If they came in the right order then print them out.

3. Otherwise, print them out in the reverse order.

Try the following program:

```
10 REM this program demonstrates the relational operators for
words (strings)
20 INPUT a$,b$
30 IF a$ < b$ THEN 60
40 PRINT b$,a$
50 GOTO 70: REM this lets us exit gracefully
60 PRINT a$,b$
70 END
```

Let's go over this program a bit more carefully. Following the THEN statement is the number 60. ADAM knows this means it should go to line 60 if the first clause was true. (You can always write the longer version THEN GOTO 60.) Suppose the word in the box *a$* didn't come before the word in

the box *b$* in alphabetical order. Then ADAM goes to line 40, which prints them in reverse order. That is what we want. ADAM then goes to line 50, which sends it to the end. What would happen without line 50? Well, we would "fall into" line 60, which would print the names in the wrong order right after we printed the names in the right order. To see this happen, try to run the program without line 50. (Actually this program is a bit clumsy. We'll show you a better way to do it later.)

Let's try to write a program that will find the smallest of three numbers. Here is the idea:

1. Enter the three numbers.

2. Set up a place for the smallest one.

3. Keep on comparing. Each time something is smaller throw it in the box holding the smallest.

Here's the program.

```
10 REM  this program lets you find the smallest of three
numbers
20 INPUT "Your three numbers please"; a,b,c
30 LET s = a: REM we'll start off with a as the smallest
40 IF b < s then let s = b
50 If c < s then let s = c
60 PRINT "the smallest number was "; s
70 END
```

Self-Check 3

A. Question

Why won't the following version of the above program work? This shows that you have to be careful when you do your translations from English to SmartBASIC.

```
10 INPUT a$,b$
20 IF a$<b$ THEN PRINT a$,b$
30 PRINT b$,a$
40 END
```

(Can you see how to fix line 30 to make this version work?)

B. Programs

Write a program that will find the smallest of four numbers; then one that finds the smallest of five numbers. (As soon as we show you a way to generate names of variables at will, this kind of program will work for as many numbers as you want.)

IF - THEN

Another use for IF-THENs is to keep a program going without getting locked into an infinite loop. It is analogous to the status check that we did in the program on page 103. In the last chapter, we wrote a program that adds any two numbers. Now let's modify it. The modified program will stop if the first number is zero. This technique is very useful; you use a ridiculous (or impossible) input to stop a program. The IF part of a ridiculous input, like a negative number for someone's salary, triggers a THEN which sends us to the exit from the program. People often use 1E38 (remember this means 1 followed by 38 zeroes) as a ridiculous input because this is about as big a number as ADAM can handle.

```
10 REM this program demonstrates a way to stop a program
20 PRINT "please enter two numbers separated by a comma"
30 INPUT "a zero 0 as your first number stops the program
"; a,b
40 IF a = 0 THEN 70
50 PRINT a+b
60 GOTO 20
70 END
```

(By the way, you must enter two numbers even if the first number was 0. Line 20 tells ADAM to expect two numbers and that's what you have to give it.)

Now let's look at a program that will calculate social security taxes. Some of you may not know about them yet, but you will! In 1984 if your salary is less than $37,800, you pay 6.7% of your salary to social security.

After that, whether you make $37,801 or $100,000, you would still pay only on the first $37,800. This works out to $2,533 for anyone whose salary is above this magic number.

The first thing we want ADAM to check is if a person's salary is more than $37,800. Such a situation is easy work for an IF-THEN. A good rule of thumb is: If you use the word *IF* in describing a situation, then a program would use an IF-THEN. Let's also arrange it so the program can keep on working on different people's salaries until someone puts in a negative salary.

```
10 REM social security tax
20 PRINT  "enter your annual salary as a number without
commas"
30 PRINT "when you want to stop, enter a negative number"
40 INPUT sal
50 IF sal < 0 THEN 90
60 IF sal >= 37800 THEN PRINT "you make more than the
 maximum so your s.s. tax is $2533"
70 IF sal < 37800 THEN PRINT "your  s.s. tax is ";sal*.067
80 GOTO 20
90 END
```

Let's examine this program to make sure we understand everything. First, lines 20 and 30 explain what we want entered when the **?** appears. Next, even though SmartBASIC only recognizes the first two letters of a variable's name, we used *sal* because it's close to the word *salary* and makes the program easier to read. Another way to make a program clearer is to use a REMark statement to explain any variable the first time it appears. For example:

```
40 INPUT sal:  REM sal stands for salary
```

The semicolon in line 70 causes the social security tax to be printed next to the explanation.

Line 80 starts the program again. This was possible because we checked for our terminator at the beginning of the program. (A *terminator* is a statement that forces the program to end.) It is a good idea to have the terminator near the beginning of a program. Otherwise you run most of the program before finding out that you didn't want to run it.

Self-Check 4

A. Questions

1. Which of the following statements are legal in SmartBASIC?

a.
```
23 IF a = yes THEN PRINT "O.K."
```
b.
```
24 IF a$<>"yes" THEN PRINT "O.K."
```
c.
```
25 IF a<10 THEN LET a = a+1
```
d.
```
26 IF a>10 THEN LET a+1 = a
```
e.
```
27 IF a$ = b THEN PRINT
```
f.
```
28 IF a$ = "b" THEN PRINT
```
g.
```
29 GOTO 50 IF a = 4
```

2. What do these programs do? Try to decide before you run them on ADAM.

a.
```
10 INPUT a$
20 IF a$ = "O.K." THEN 50
30 PRINT "YOU'RE ";a$;". THAT'S
   TOO BAD."
40 GOTO 60
50 PRINT "I'M O.K. TOO"
60 END
```

What would be a good input prompt in line 10?

b.
```
10 INPUT x
20 IF x*x<x THEN PRINT "YOUR NUMBER
   IS SMALL"
30 END
```

c.
```
10 INPUT x,y
20 IF x+y < 50 GOTO 50
30 PRINT "TOGETHER YOU HAVE ENOUGH
   MONEY"
40 GOTO 60
50 PRINT "SORRY.  I CAN'T LET YOU
   IN."
60 END
```

B. Programs

Write programs for the following:

1. A sales rep gets a one thousand dollar bonus if he or she sells a thousand widgets, but gets nothing if he or she sells less. Write a program that will allow someone to input the number of widgets he or she sold and then have the program determine if the sales rep got the bonus.
2. On some credit cards you pay 18% interest on what you owe if you owe less than fifteen hundred dollars, but you pay only 12% on the amount over this. Write a program that will let you input the amount you owe and then print out the interest for the month.
3. You may remember from high school that a quadratic equation is an equation like:

   $$ax^2 + bx + c$$

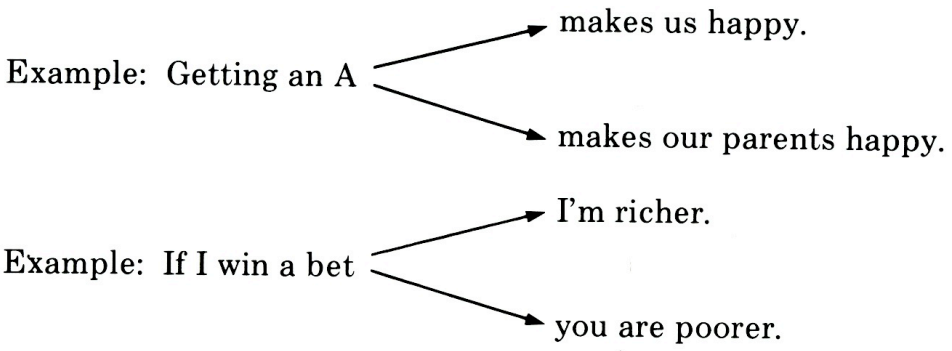
   These equations have "imaginary roots" if $b^2 - 4ac$ is less than 0.

   Write a program that will input a, b, and c and then tell you if the quadratic equation has imaginary roots. (If you've long since forgotten what imaginary roots are, don't worry—just forget this problem.)

Other Uses of IF-THEN

What other ways can we use IF-THEN? As we've mentioned, the analogy with normal speech is a good one. Below is a short list of ways that we use the IF-THEN.

1. Do two things if a single condition is true.



2. When two things are true we do a single thing, but if either one of them fails we do nothing.

   Example: If we have (1) a sunny day, AND (2) a certain temperature, then we can sunbathe.

3. If either of two things happen, we do a third.

   Example: If it's lunchtime or dinnertime then we can eat.

4. If something doesn't happen, then we do something.

   Example: If it isn't dark, then we can play basketball.

Each of the above situations has its counterpart in SmartBASIC. Let's discuss them in order.

First, doing two things when a single (good) thing happens. Here is the first part of a program that a company might use to calculate both the discount a purchaser receives and the bonus its sales rep got for taking a large order.

```
10 REM This program demonstrates the use of the colon
20 LET bonus = 0: LET discount= 40
30 INPUT "How many widgets did you sell?"; n
40 IF n >= 1000 THEN bonus =1000: discount= 50
                    .
                    .
                    .
```

There isn't anything odd about this kind of situation. We just want to do two things when our sales rep sells more than one thousand widgets in any order. Another place we could have used the colon was in the program to rearrange two words in the correct alphabetical order. You may have noticed that there was a sort of awkwardness in lines 30 and 50. Look at the following version:

```
10 REM 2 for the price of 1
20 INPUT a$,b$
30 IF a$ < b$ THEN PRINT a$,b$: GOTO 50
40 PRINT b$,a$
50 END
```

Line 30 is crucial. Let's look at it more carefully. Suppose the first word did precede the second. Then what we really wanted to do was print them out in the same order they were entered and end the program. The colon lets us do this. Whenever ADAM encounters a colon after having successfully tested the first part of an IF-THEN, it understands this to mean "Do all the statements on that line, no matter how many statements separated by colons are on the line." Again, be careful here. Line doesn't mean a line on your television; it means the logical line, everything that ADAM encounters before the next numbered line. (In the example above, the next numbered line is line number 40.) Besides making the IF-THEN a far more powerful programming tool, a colon often makes a direct translation from English easier. Also, unless overdone, it usually makes your programs easier to read.

The idea of checking two things before you do a third is pretty commonplace. For example, some new cars have microcomputers checking that the number of revolutions of the motor each minute is within some critical range. We can mimic this kind of check using the following program fragment:

is a perfectly good SmartBASIC statement. In the Self-Checks there will be many times when this type of statement is needed.

How does ADAM mimic the third use of IF-THEN? We are using the logic of: if something or if perhaps something else, then . . . . So naturally SmartBASIC uses the word OR, as in the following program:

```
10 REM A demonstration of the OR
20 PRINT "Type one upper case letter then hit return"
30 PRINT "I'll tell you if it's a vowel"
40 INPUT le$
50 IF le$ = "A" OR le$ = "E" OR le$ = "I" OR le$ = "O" OR le$
   = "U" THEN PRINT "your letter was a vowel": GOTO 80
60 IF le$ = "Y" THEN PRINT "Y is too hard for me" : GOTO 80
70 PRINT "Your letter was a consonant"
80 END
```

Notice that ADAM had a lot of checking to do on line 50. We actually didn't have to use any ORs. We could have written something wasteful of time and energy like:

```
41 IF le$ = "A" THEN. ...
42 IF le$ = "I" THEN. ...
```

and so on.

Also note that we use the colon to keep ADAM from falling into PRINT statements that would print incorrect sentences.

We don't find much use for the word NOT. It's usually easier to just change symbols. For example, the clause:

```
IF NOT (a$ = "ADAM")
```

is harder to write than:

```
IF a$ <> "ADAM"
```

Similarly:

```
IF NOT(a > 50)
```

is exactly the same as typing:

```
IF a <= 50
```

If you prefer to use the NOT, you'll need to use parentheses.

Trick of the Trade There is one use of the NOT command that comes up in advanced BASIC programming. To understand it, you have to know that, in reality, computers only deal with numbers. This means that ADAM needs a way to understand true and false statements in terms of numbers. To ADAM "TRUE" is represented by the number 1 and "FALSE" by the number 0. This leads to the following useful tool: suppose you have a variable *swt* that you want to frequently change between values 0 and 1. (This imitates turning a switch on and off.) You just use the command:

```
swt = NOT(swt)
```

This command changes the current value of *swt* to 0 if it was 1 and to 1 if it was 0. This same trick can be used with the AND and OR commands

The idea that computers use 1 to represent truth and 0 to represent falsehood is also used in IF-THENs. For example, in direct mode enter:

```
LET test = 1
IF test THEN PRINT "true"
```

The two statements will always tell ADAM to display the word "true". The IF-clause has value 1, so the statement in the THEN-clause is executed. If the first line was:

```
LET test = 0
```

nothing would be displayed because the IF-clause is interpreted as being false.

Self-Check 5

A. Questions

1. Are the following statements legal in SmartBASIC?

   a.
   ```
   12 IF a>3 AND <4 THEN PRINT "A IS
      SMALL"
   ```

   b.
   ```
   13 IF a>b AND b<4 THEN PRINT "A IS
      4 ALSO"
   ```

   c.
   ```
   14 IF a<5 OR b<9 THEN PRINT
   ```

   d.
   ```
   15 IF a<5 THEN PRINT:  PRINT "HI
      THERE"
   ```

   e.
   ```
   16 IF NOT(a<=0) THEN PRINT "A WAS>
      0 "
   ```

2. What do the following programs do? (Think about them before you try them on ADAM.)

   a.
   ```
   10 INPUT a,b
   20 IF a < 0 OR B < 0 THEN 40
   30 GOTO 10
   40 END
   ```

   b.
   ```
   10 INPUT a$,b$
   20 IF a$="no" AND b$="no" THEN 40
   ```

(continued)

```
30 GOTO 10
40 PRINT "I'M DONE"
50 END
```

Keepting Count: LOOPING

We can use an IF-THEN to keep count. Suppose someone has a list of names and just wants to know how many names there are on the list. Obviously you could count them and make a tally sheet with one mark for each name.

Let's write a program to do this. We need a box to keep our count (we will call it *cnt*) and each time we read a name, we put a mark in the box. This means we add one to the total that was in the tally box named *cnt*. This process is called *incrementing* a counter—if we were counting down, the word would be *decrementing*. Finally, we will stop counting when zzz is entered.

```
10 REM This program demonstrates the use of a counter
20 REM It's one of the most useful ideas in programming
30 LET cnt = 0: REM start the counter from zero
40 INPUT "Type one name and hit return each time the ?
appears.  To stop the counter type in zzz. "; name$
50 IF name$ = "zzz" THEN 80
60 LET cnt= cnt + 1: REM the turnstile spins
70 GOTO 40 : REM get another name
80 PRINT "Your list had ";cnt;" names"
90 END
```

Let's go over this program line by line.

Lines 10 and 20 just explain the program.

Line 30 sets the counter to 0. Notice that we have explained the line by using a REMark statement.

Line 40 uses an INPUT prompt to explain what we want when the **?** appears.

Line 50 checks if ADAM has encountered a zzz and, if so, it sends ADAM to line 90 and ends the run.

Line 60 throws another 1 in the counter. Notice that we have put this line after the test. That's because if the first name was a zzz, we would not have any names and so the counter should be 0.

Line 70 sends us back for another name.

A common error in line 70 is to write GOTO 30. What would happen if we wound up on line 30? Line 30 resets the count to 0. The PRINT statement would always tell us that the list had no names on it.

Line 80 prints the final total. Notice the only way to get to this line is from GOTO on line 50. The other GOTO on line 70 acts as a barrier and there is no other route to line 90.

You may be wondering if there is a way to save the names as well as count them. To save them, we have to have some place to put them, like the filing cabinet we saw in chapter 2. We'll show you in chapter 16 how this is done using files.

What we are doing in these programs is best seen by looking at Figure 5.2.

This is one of the most powerful techniques in programming. It lets you do something as many times as you want and then stop. A good example to keep in mind would be a program to find out how many years it will take your money to double at a specified rate of interest. Since you don't know beforehand what the answer is, you have to keep on compounding the interest until you have at least twice the amount you started with. Can you see how to write the program? Here is an outline:

1. Get the amount and the interest rate.

2. Start computing the interest you get at the end of each interest period.

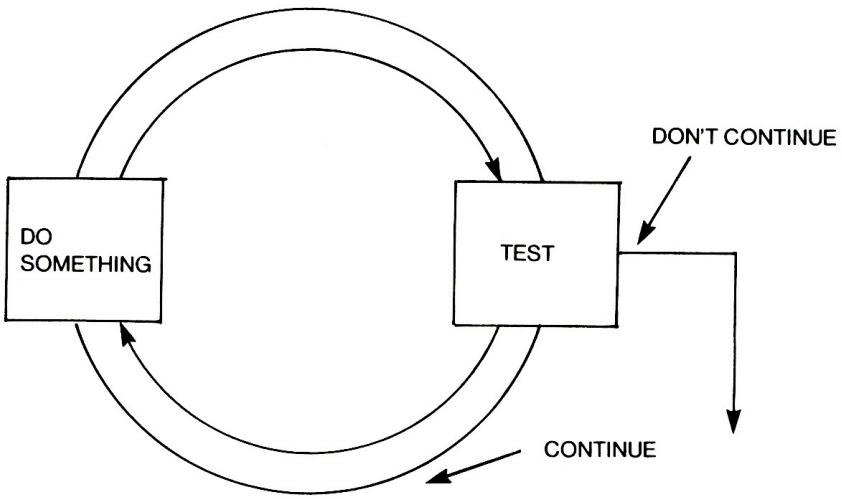


Fig. 5.2 Calling out of a loop.

3. Add the interest to the old amount.

4. Is this new amount at least twice the original amount? If it is, print out the number of times you did step 2.

5. If it's not yet doubled, go back to step 2.

Here is another example. Everyone knows what an average is, but if you follow the stock market, you may have heard the expression "moving average." In a moving average, you average more and more numbers. Each time you enter a new number you compute a new average including that number. Let's write a program to compute a moving average. Here is a description in English:

1. Get a number.

2. Add it to the previous sum.

3. Find the new average by dividing the new sum by one more than whatever the number of things you have already added.

Here's the SmartBASIC program:

```
10 REM a moving average
20 n = 0: REM n counts the number of numbers
30 sum = 0: REM the sum starts at 0
40 INPUT x
50 LET s = s+x: REM new sum is x more then old
60 LET n = n+1: REM one more number
70 LET av = s/n
80 PRINT "the moving average is "; av
90 GOTO 40
100 END
```

(To stop this, you will have to type CONTROL/C, unless you can think of a terminator.)

Punishment Program Revisited

We finally have enough commands to write that punishment program we spoke of at the beginning of this chapter. What follows isn't the easiest way to do it, but it illustrates an important method. We will soon show you another command that makes it much easier. We hope that you will try to analyze it, because the ideas are important.

```
10 REM Here is the punishment program
20 LET cnt = 1
30 PRINT "I will not be late"
40 IF cnt = 500 THEN 70
50 LET cnt = cnt + 1
60 GOTO 30
70 END
```

Self-Check 6

A. Questions

1. Does the following program have a terminator?

```
10 REM this program computes your
   average monthly salary
```

(continued)

```
20 INPUT "INPUT YOUR WEEKLY SALARY";ws
30 INPUT "INPUT YOUR CHRISTMAS BONUS";bonus
40 IF c = 0 THEN 70
50 PRINT ((52*ws)+bonus)/12;" IS YOUR AVERAGE
   MONTHLY SALARY"
60 GOTO 10
70 END
```

2. Would the above program have a terminator if you add the following line?

```
25 LET ws = a
```

3. Would the program in 1 above have a terminator if, instead of adding line 25, you add in the line:

```
15 ENTER -1 FOR WEEKLY SALARY TO EXIT
```

4. Rewrite the line directly above as a statement in SmartBASIC and retry the question.
5. Why not use 1E38 as a terminator for the moving average program? Be careful about where in the program you test for the terminator.
6. Would our final punishment program work if the statements on lines 40 and 50 were interchanged?

B. Programs

1. Write a program that will print the numbers from 1 to 20.
2. Write the program to do the interest rate calculation.

Summary

In this chapter you learned how to change the order in which ADAM processes your program lines. We showed you how to have ADAM ask questions using these relational operators:

=	<>
<	<=
>	>=

and how to use the following commands:

GOTO	Sends ADAM to a specified line number.
ON __ GOTO __	To use a menu-driven program.
IF-THEN	To have ADAM ask questions and check (test) things.
CONTROL/C	To stop a program.
CONT	To start a program that has been stopped.

Finally, we showed you how to make the ADAM's tests more interesting and powerful using AND, OR, NOT, and colons.

BUZZwords

BRANCHING

BREAK(ING)

DECREMENT

EXECUTED

INCREMENT

INFINITE LOOP
(ENDLESS LOOP)

INTERRUPT

LOOPING (LOOP)

MENU-DRIVEN

MODULAR PROGRAMMING

RELATIONAL OPERATORS

TERMINATOR

TRAP

USER FRIENDLY



Doing Things Repeatedly

In this chapter we will show you another way to program repetitive operations. You'll learn the following commands in SmartBASIC:

FOR	NEXT	STEP
PR#1	PR#0	CONTROL/S
SPEED		

Repeated Operations – A Second Look

After working through the punishment program at the end of the last chapter, it's natural to wonder why SmartBASIC doesn't just let you say:

```
DO -----500 TIMES
```

where you can fill in the blank with whatever SmartBASIC commands you want. It's easy to see how such a command would make your translations from English to SmartBASIC more straightforward. Before we show you how this is implemented (*implemented* is a buzzword for how something is done) in SmartBASIC, let's write down, in English, some other commonly used repetitive operations.

1. Compute the sales tax on different dollar amounts.

2. Give a conversion table for degrees Celsius to degrees Fahrenheit.

3. Compute a mortgage payment table where the payment stays the same but the interest rate changes.

4. Compute income tax tables.

The people who invented BASIC wanted a very general command that would allow BASIC to perform many different types of repetitive operations—including printing "I will not be late" five hundred times and doing the fourth computation listed above. The command they decided to use is similar to a command which exists in FORTRAN. It looks like:

```
FOR

( whatever you want done
   in SmartBasic )

NEXT
```

FOR and NEXT are a pair of commands. The part of the program starting with FOR and ending with NEXT is called a *FOR-NEXT loop.* (Remember that *loop* is the buzzword for something done repeatedly.) Using a FOR-NEXT loop, we can write the punishment program as:

```
10 REM a demo of the FOR-NEXT LOOP
20 FOR i = 1 TO 500
30 PRINT "I will not be late"
40 NEXT i
50 END
```

Think of a FOR-NEXT loop as winding up a wheel inside ADAM so that it spins a certain (fixed) number of times. At each spin of the wheel, ADAM can be told to do something.

The command FOR sets up the *counter variable, i,* and the command NEXT changes the counter. Between the commands FOR and NEXT is the part of the program called the *body* of the loop, which consists of various statements in SmartBASIC. Every time ADAM reaches the word NEXT, it adds one to the counter variable and then checks (or tests) to see whether the counter variable, in this case the variable *i,* has gone past the test value. (*Test value* is the buzzword for the last number on the line starting with FOR.) If the counter variable hasn't gone past the test value, ADAM goes back to the first statement following the FOR. The cycle starts again and only stops when the counter variable is past the test value. Because of this any FOR-NEXT loop runs at least once.

You can use any numeric variable name for the counter variable. Most people like to use the letter *i* to remind them of the word *index,* which is another word for counter. We will use whatever variable name seems to best suit our use.

Here is another example:

```
10 FOR a = 1 TO 10000
20 PRINT "$";
30 NEXT a
40 END
```

This program prints ten thousand dollar signs next to one another until you run out of space on a line on the screen, then it scrolls (jumps) to the next line. If you ran it, you might want to stop it by a CONTROL/C (see the previous chapter).

We haven't yet explained why the inventors of BASIC decided to use this slightly more complicated and certainly stranger way to set up a wheel than the simple "DO SOMETHING" that we mentioned above. The reason is that the variable following the command FOR may be used in the body of

the loop. Most often you don't want to do exactly the same thing time after time. Something changes. We use the counting variable to grab hold of that change. For example, to print the numbers from 1 to 100, we could use the following program.

```
10 FOR n = 1 TO 100
20 PRINT n
30 NEXT  n
40 END
```

In this program the counter variable appears in the PRINT statement in the body of the loop.

The possibility of using the variable *n* both as the counter variable and inside the loop makes this program as short as possible. Once you get used to this command you will find that the strangeness fades and programs that have repetitive operations are easy to do using FOR-NEXT loops. The only thing to be aware of is that it is a very bad idea to modify your counter within the loop. If you change its value by means of an assignment (LET) statement, your program will almost always not do what you want. The following (bad) example is worth thinking about:

```
10 REM a very bad program
20 FOR t = 1 TO 100
30 LET t= .07*I
40 PRINT t
50 NEXT t
```

The programmer was trying to calculate a sales tax table for a 7% tax but forgot that he or she was also using *t* as the counter. The result (try it) is a program that does something very strange. If you like mathematical puzzles, try to figure out what is happening.

Suppose you really wanted to prepare a sales tax table in a state with a 7% sales tax, for amounts from $1 to $100. The outline is quite simple.

Find the sales tax on $1

Find the sales tax on $2

.

.

.

Find it on $100.

Here's the program:

```
10 REM a simple sales tax table
20 PRINT "            SALES TAX TABLE "
30 PRINT:PRINT
40 PRINT "SALES PRICE","TAX"
50 PRINT:PRINT
60 FOR p =1 TO 100  :REM p is the dollar amount of the sale
70 PRINT "$";p,"$";.07*p
80 NEXT p
90 END
```

Let's go over this program carefully.

Line 20 is used to print the heading for the table. We used spaces within the quotations so that the heading would be centered.

Line 30 prints two carriage returns (notice the colon).

Line 40 prints the words SALES PRICE and then uses a comma to force the tax to be printed near the middle of the screen (in the next print zone).

Line 50 is just like line 30.

Line 60 begins our loop. It tells ADAM to do something one hundred times.

Line 70 is where everything happens. We use the quotes to print a dollar sign next to the price, then ADAM printed the price. Following that ADAM encounters the comma, which forces it to print the next thing in the middle of the screen. So that's where it prints the sales tax on the amount of the sale. Notice that none of the actions taken by this statement changes the counter *p*.

Line 80 has the crucial command NEXT. It does two things.

1. Increases the counter variable named *p* by one. This puts us at the next dollar amount for the sales price.

2. Sends you back to line 70 if the test value has not been exceeded.

Line 90 is the END statement. ADAM only processes line 90 when it has finished repeating the operations in lines 60 to 80.

One last example: Suppose you wanted to find out the product of the numbers one through fourteen. (If you are wondering why anyone would want to know that number, it's the number of different possible orders in which teams in baseball's American League can finish.) The outline is clear: multiply 1 by 2, 2(=1*2) by 3, 6(=1*2*3) by 4 and so on. Here's the program:

```
10 REM the product of the numbers one through fourteen
20 LET ans = 1
30 FOR s = 1 TO 14
40 LET ans = ans * s
50 NEXT s
60 PRINT "the answer is  ";ans
70 END
```

Do you see why we have to initialize (*initialize* is the buzzword for start) the variable *ans* with the value 1? Try this program without line 20. If you run it, you'll get 0 as the answer. In SmartBASIC the first time a variable is mentioned, ADAM gives it the value zero unless you initialize it to some other value.

The product of the numbers from one to n is often needed in mathematics. It's used whenever you try to calculate the number of different ways various things can happen. It's called "n factorial" and the symbol used is **n!**, so what ADAM computed was 14!

After you outline your program, there is a good rule of thumb for deciding if you might need a FOR-NEXT loop. If you say to yourself something like "perform an operation a certain number of times," you'll need either a FOR-NEXT loop or a program similar to the ones at the end of the last chapter.

Self-Check 1

A) Questions

1. What's wrong with the following program?

```
10 FOR i = 1 TO 500
20 PRINT "I WILL NOT BE LATE"
30 END
```

2. What does the following program do?

```
10 FOR i = 1 TO 100
20 PRINT i-1
30 NEXT i
```

B) Programs

1. Write a program that will print the numbers from 1 to 20.
2. Write a program that will find the product of the first 30 numbers (the answer is quite large).
3. Write a program that will print the first sixty-four powers of two.
4. Write a program that will add the first ten numbers, the first hundred numbers, the first thousand. Do you see a pattern?

5. Often when you go to a store that is having a sale, there will be lists of the original prices and the discounted prices. Write a program that will let a shopkeeper tell you the discount and then print out a table of original prices from $1 to $100 and the corresponding discounted prices.

If you did any of the Self-Checks, especially the fourth, you might be wondering if there is any way of changing the counter at the beginning of a program. This isn't hard in SmartBASIC because you can use a variable instead of a number to indicate where the counter variable starts and ends. (It should come as no surprise, that this starting point has *starting value* for its buzzword. As we mentioned before, the buzzword for the end value is *test value*.) Here is a program that will let the person sitting at the keyboard decide how many numbers to print.

```
10 REM a variable test value in a FOR-NEXT
20 INPUT "How many numbers ? ";n
30 FOR i=1 TO n
40 PRINT i
50 NEXT i
60 END
```

Not only can you use the value of a variable (remember the value is the contents of a box) in the FOR-NEXT loop, you don't have to start the counter at 1. You could start it at −37, or at a different variable, or even at 42 times some variable. For example, you may have noticed that, to compute the product of the first 14 numbers, we didn't have to use one as the starting value of the counter variable. It would have made a lot more sense to do the following:

```
10   REM the product of the first 12 numbers done right
20   LET ans = 1
30   FOR n= 2 TO 14
40   LET ans=ans * n
50   NEXT n
60   PRINT "14! = ";ans
70   END
```

Here's another example. Suppose we wanted to print a table of conversions from Celsius to Fahrenheit. However, let's allow the person sitting at the screen to decide where to start and where to end. Here's the program:

```
10 REM a temperature conversion table
20 HOME: PRINT "What are the ranges for the table?"
30 INPUT "Enter the lower number, a comma and then the higher
number ";lo,hi
40 PRINT  "CELSIUS", "FAHRENHEIT"
50 PRINT:PRINT:PRINT
60 FOR cel = lo TO hi
70 LET F = (9/5)*cel +32: REM  this does the conversion
80 PRINT cel,F
90 NEXT cel
100 END
```

Don't run it yet. First let's discuss it. Lines 20 and 30 just clear the screen and prompt the operator.

Lines 40 and 50 set up the table.

The variable *cel* in line 60 is supposed to remind us of the word Celsius. (You could even use the word *celsius* as the counter if you are better at typing than we are. We think *cel* is a good compromise between just using a *c* and using the whole word.)

Line 70 calculates the temperature in Fahrenheit. It could have been done in the PRINT statement on line 80, but we don't like long and cluttered program lines. It's easier (and safer) to do your calculations on separate lines. Notice we are taking advantage of ADAM doing addition as the last operation. This lets us avoid yet another set of parentheses. (If you studied the examples in chapter 4, you know that we didn't need any parentheses at all. But old habits die slowly and the formula we used, parentheses and all, is the one we learned in school.)

Now it's time to run the program. It really goes fast—perhaps too fast to read. How can you stop a display to give yourself a chance to read it?

> ADAM LETS YOU FREEZE A DISPLAY BY HOLDING DOWN THE CONTROL KEY AND TYPING AN S. (WE'LL CALL IT CONTROL/S.) YOU CAN RESTART THE DISPLAY BY HITTING ANY KEY.

When you use CONTROL/S, ADAM temporarily stops running the program and waits for you to hit any key before continuing.

While we are on the subject of things going by too fast, you may have noticed that, sometimes at least, it would be nice to have ADAM slow down. There are two ways of slowing down the computer. One slows down printing and the other actually slows computation. We'll do both in this chapter.

First, let's suppose you want to display a message on the screen for a few seconds, erase it, then display another message, and so on. You can do this by a trick which uses a FOR-NEXT loop. Remember that when ADAM encounters a FOR NEXT loop, it must cycle (or spin a wheel) at least as many times as you tell it. While ADAM is very fast, each spin takes a certain amount of time (about 1/1000 of a second). So a program fragment like:

```
          .
          .
          .
1000 FOR pause = 1 TO 1000
1010 NEXT pause
          .
          .
          .
```

will stop ADAM from executing any further instructions until it finishes spinning the wheel a thousand times. As we said, ADAM will spend about one second executing this loop—which does nothing but kill time. But why do we want to just kill time? Well, suppose you want to give people a chance to see how fast they can read something. Using a timing loop (*timing loop* is computerese for a loop that kills time), we can write a program whose outline is the following:

1. Print something

2. Wait a bit

3. Erase it

4. Print again

and so on.

Here is a part of the program (it's only a part because we will soon show you a much better way to do it):

```
10 REM a demonstration of timing loops
20 HOME
30 PRINT "         CAN YOU READ THIS"
40 FOR p = 1 TO 500
50 NEXT p
60 HOME
70 PRINT "              OR THIS"
80 FOR p = 1 TO 1000
90 NEXT p
100 HOME
110 PRINT "           TRY AGAIN"
120 FOR p = 1 TO 2000
130 NEXT p
140 HOME
        .
        .
        .
```

The only thing that may seem odd in this program is that we kept using *p* as our variable. There is nothing wrong with using the same counter variable here because each time you start a FOR-NEXT loop, ADAM starts the counter at the value you set regardless of its previous value.

You can keep on writing as many timing loops as you want. Just remember that each thousand added to the test value will add about one second. So a timing loop with $p = 10000$ will take about ten seconds. (We say about because as the numbers get bigger, it's not quite one second per thousand. The difference is minor and due to some internal characteristics of ADAM. It isn't worth worrying about.)

Here's a slightly different version of a reading program. It allows the reader to decide how much time he or she needs.

```
10 REM a reading test
20 HOME
30 PRINT " When the question mark appears"
```

(continued)

```
40 PRINT " please enter the number of"
50 PRINT " seconds that you think you"
60 PRINT " will need to read the"
70 PRINT " Preamble to the United States"
80 PRINT " Constitution"
90 PRINT " Hit CONTROL/C when you're done"
100 INPUT sec
110 HOME
120 PRINT " We, the People of the United"
130 PRINT " States, in order to form a"
140 PRINT " more perfect union, estab-"
150 PRINT " lish justice, insure domestic"
160 PRINT " tranquility, provide for the"
170 PRINT " common defence, promote the"
180 PRINT " general welfare, and secure"
190 PRINT " the blessing of liberty to"
200 PRINT " ourselves and our posterity,"
210 PRINT " do ordain and establish this"
220 PRINT " Constitution of the United"
230 PRINT " States of America."
240 FOR p = 1 TO 1000*sec
250 NEXT p
260 HOME
270 PRINT " You guessed too low"
280 END
```

We used many PRINT statements and extra spaces. By doing so, we prevent words from falling off the screen due to overscan (see chapter 1). ADAM prints exactly thirty-one characters per line and then continues on the next line. Carefully laying out the printed text can prevent words from being "chopped up" in weird ways. This program is a good test (and not only of speed-reading) that you might want to try on some friends. The only line in the program that we want to explain is line 240. ADAM will usually do any calculations called for in a command before executing the command. We have already seen this in the PRINT command. If we entered:

```
PRINT 1000*3
```

ADAM first did the calculation and then printed the result. On line 240, ADAM calculates the test value by multiplying 1000 by the value of the variable *sec.*

Trick of the trade: You don't need to say NEXT *p* or, in general, NEXT *counting variable.* ADAM keeps track of the name you gave to the counter. For this reason you will often see a timing loop done on one line like

```
FOR p = 1 TO 5000:  NEXT
```

A timing loop is speeded up by about 10% by using the command NEXT without specifying the variable. Other types of FOR-NEXT loops run faster when the counter variable is not specified.

In a FOR-NEXT loop, ADAM really does change the counter. You can see this by running the following program:

```
10 FOR p = 1 TO 1000
20 NEXT p
30 PRINT "I'm done. What's next?"
40 END
```

When you see the SmartBASIC prompt], type PRINT p. The screen should show:

```
] print p
```

Now hit RETURN and ADAM will print out:

```
1001
```

ADAM has systematically been adding one to the value of *p* and then it has been going back to check whether it has gone past the test value you have placed on *p* (in this case 1000). The first time that it sees that it has gone past the test value it immediately goes to the statement following the NEXT statement. The value of *p* is stuck at 1001. You can always check what values your variables have by waiting for the] and telling ADAM to print them.

With a FOR-NEXT loop you can compute compound interest without ever knowing a formula. Instead let's use some common sense and our knowledge of SmartBASIC. What happens in compound interest? You get

your interest added to your original amount to form your new amount. Then you take the interest on this new amount and add it and keep going. If you stop for a second and think about what we just said you'll realize that we've just given you an outline for the program.

1. Get the original amount.
2. Find out how many times to compound it.
3. Find the interest on it.
4. Add it to the original amount.
5. Go back to step 3 as many times as you are doing the compounding.
6. When you're finished, print out the final amount.

You need to remember one thing: if you make, say, 12% interest per year compounded quarterly, you get 3% interest each quarter year. In general, find the interest rate for an interest period you divide the annual interest rate by the number of times per year that interest is compounded. The interest rate for a period is the number you use in step 3.

```
10 REM compound interest without formulas
20 HOME
30 INPUT "What's the amount you started with? "; amt
40 INPUT "What's the interest rate per year? "; int
50 PRINT "How many times is it to be compounded?"
60 PRINT "For example for quarterly, type in a 4"
70 INPUT p: REM p is the # of interest periods
80 REM we start the compounding here
90 FOR n = 1 TO p
100 LET mon = (amt*int/p): REM mon is the interest we get at
any particular time
110 LET amt = amt + mon: REM the new amt is the old + mon
120 NEXT n
130 PRINT amt
140 END
```

Each time you write a piece or module of the program that does a specific job you should explain it. We used a REMark as line 80, in the middle of a program. This helps people—including yourself—to read your

programs. (By the way, you may have noticed that lines 100 and 110 could have been put together; we didn't because we were trying to follow the outline.)

Self-Check 2

A. Question

About how many seconds will the following program run?

```
10 FOR p = 1 TO 26550
20 NEXT p
30 PRINT "That's long enough."
40 END
```

B. Programs

1. Write a program that will add the numbers from one up to a number that has been typed in by the person running the program.

2. Inflation is just like compound interest in reverse. Write a program that will compute how much one thousand dollars will be worth in twenty years. The operator should be able to type in some interest rate in response to an INPUT prompt.

3. Suppose you deposit one thousand dollars on January 1 in a bank account. Each year you get 10% interest, compounded two times during the year, on what you have in the account. Write a program that will figure out how much money you have after twenty years and after thirty years.

Doing it by Steps

You don't always count things one by one. Sometimes its easier to count by twos or threes, or even backwards. You can do this in SmartBASIC by using a STEP command in your FOR-NEXT loop. The STEP command tells ADAM to change the counter by the STEP amount rather than by one. Here's an example:

```
10 REM using a step - the odd numbers
20 FOR i = 1 TO 100 STEP 2
30 PRINT i
40 NEXT i
50 END
```

Each time ADAM gets to line 40, it adds 2 rather than 1 to the value of *i*. The program prints the odd numbers from 1 to 100. Do you see why it stops at 99?

You can also count backwards by using a negative STEP. For example, here is a "blast-off" program.

```
10 REM a negative STEP
20 FOR t = 10 TO 1 STEP -1
30 PRINT "It's T minus "; t ; " and counting"
40 NEXT t
50 PRINT "We have liftoff!"
60 END
```

(Don't blame us for this strange language. It's not computerese. It's straight from the space program.)

If you use a negative STEP you should make sure that the first number is bigger than the second; otherwise ADAM makes only one pass through the loop. The same thing happens if we have positive steps and the starting value is not smaller than the test value.

The STEP command is very useful. Suppose you wanted to have a handy reference chart for changing degrees Celsius to degrees Fahrenheit.

In order to keep it short, we'll only have ADAM give the conversion for five degree changes. Here's a program that will do it for you.

```
10 REM a handy temperature conversion table
20 PRINT "   CELSIUS", "FAHRENHEIT"
30 PRINT:PRINT:PRINT
40 FOR cel = -40 TO 100 STEP 5
50 LET f = (9/5)*cel +32: REM  this line does the conversion
60 PRINT cel,f
70 NEXT cel
80 END
```

Each PRINT statement will direct the output of the program to the television screen. You might also like to keep a permanent copy, that is, a paper copy. To get a hard copy (*hard copy* is computerese for paper copy), you can try CONTROL/P, but it only prints a "photograph" of the contents of the screen. You can't really get a table printed this way. Here's where ADAM proves that it is useful to package a printer with a computer system. All you need do is learn one more command:

```
PR#1
```

Here is how to use it in the program above. Add a line 15 (you can see why increasing line numbers by 10 is useful) that says:

```
15 PR#1
```

That's all; when you run the program you get your hard copy on the printer.

One thing to be aware of is that you must give control back to the screen—otherwise, from the time that the PR#1 statement is read by ADAM, everything printed on the screen will also be sent to the printer. You can waste an incredible amount of paper this way. There are three ways to cancel the PR#1 command. You can:

1. Type PR#0 after the program ends; or (better),

2. Right before the END command, add a line with the command PR#0.

3. CONTROL/C can be used to cancel PR#1.

Tricks of the trade: Something you often want to do is get a hard copy of a program listing. You can do this by typing, in direct mode:

```
] PR#1:LIST:PR#0
```

and hitting RETURN.
This will get you your hard copy and leave you back with the screen as the output device. It also shows you that, even in immediate (calculator) mode, you can do more than one thing in a single line.

Here is another example of where using PR#0 and PR#1 proves handy. Suppose we want to send out 100 change of address announcements. Since this is the age of personalized form letters, we want to have each one individually addressed.

```
10 REM a form letter program
20 FOR n = 1 TO 100
30 INPUT "WHAT NAME? "; name$
40 INPUT "WHAT ADDRESS? "; ad$
50 PR#1
60 PRINT name$
70 PRINT ad$
80 PRINT:PRINT
90 PRINT "    My new address is 1313 Mockingbird Lane.  I'm
having a house-warming party Friday night the 13'th.
Please come."
100 PR#0
110 NEXT n
120 END
```

Lines 10–40 just allow you to input the names and addresses.

Line 50 will send any future PRINT statements to the printer.

Line 90 takes up more than one line on your screen but the printer can handle eighty characters on each line, so the format on the screen has little to do with the format of the hard copy you get. Finally, before ADAM ends one cycle of the loop, the command PR#0 disables the printer. (*Disable* is the buzzword meaning shut off.) The printer should be disabled while we are responding to the INPUT requests. Otherwise the printer would also type out these requests.

> THE COMMAND PR#1 SENDS ADAM'S OUTPUT TO THE PRINTER IN ADDITION TO THE SCREEN. THE COMMAND PR#0 UNDOES PR#1; i.e., IT DISABLES THE PRINTER.

To this point we've been using ordinary (counting) numbers for our counters and our STEP. This isn't necessary; you can start anywhere and step up (or down) by any amount that is convenient. For example, many mortgage tables are arranged by ¼% amounts.

You may have heard the story that the Indians sold Manhattan Island to the Dutch for $24 worth of trinkets in 1625. Have you ever stopped to think how much $24 would be worth now? We shoud find out the results for a range of different rates of interest, since we really don't know what rate is appropriate. Let's write a program that gives a table of current values of the $24 assuming different rates of interest.

To do this problem, we will use the formula for compound interest that you probably learned in school. Of course, after the first section of this chapter, you don't need it, but then your program would become a lot more complicated. There is always a tradeoff—you can memorize a lot of formulas or use some special trick that will make your programs shorter. However, keep in mind that the use of special tricks may make programming much harder to write and debug. If you didn't learn the formula in school, it's easy enough just to accept it.

We start off with an initial amount *p,* called the principal, and a certain interest rate, *i.* After a certain number of interest periods which we will call *pe,* the amount you end up with (we'll call it *amt*) is:

$$\text{amt} = p*(1+i) \wedge pe$$

where ^ is the exponentiation symbol described in chapter 3.

```
10 REM Was Manhattan a good buy?
20 PRINT " The Indians sold Manhatten for $24 in 1625"
30 PRINT " Here is a table of what that $24 would be worth
today at various interest rates"
40 PRINT:PRINT
50 PRINT "INTEREST RATE","    AMOUNT"
60 PRINT:PRINT
70 FOR i = .01 TO .15 STEP .01: REM percents are decimals
80 PRINT "    ";i*100;"%","$";24*(1+i)^359
90 NEXT i
100 END
```

Lines 20–60 just format the table.

Let's go over the crucial lines, 70 to 90. We decided to do the table at interest rates starting at 1% (decimal .01) and going to 15% (decimal .15). The steps go up by 1% which is .01. Notice that by multiplying *i* by 100 we get the percent, but we must use decimals because percent is not a concept recognized by SmartBASIC. Line 80 prints out the results. Then the semicolon tells ADAM to print the percent sign next to it. The comma will move things to the next column to make the display look neat.

If you run the program, you will find that the value of Manhattan Island today—land, skyscrapers, and all—is considerably less than $24 compounded at 8%. It is unlikely that anyone could have received interest compounded at 8% for so long.

Self-Check 3

A. Questions

1. How many times would the following FOR-NEXT loops run?
   a. FOR j = −1 TO 1 STEP 2
   b. FOR j = 1 TO 2 STEP 2
   c. FOR j = 3 TO 2 STEP 2
   d. FOR i = 3 TO 2 STEP 2
   e. FOR p = 1 TO 5 STEP −1
2. In the following program, what are the test value, the starting value, and the step?

```
10 LET a = 3:  LET b = 4:  LET c = 15
20 FOR i = a TO c STEP b-a
30 NEXT i
```

B. Programs

1. Write a program that will add the first ten odd numbers, the first twenty odd numbers, and then the first hundred. (Do you see a pattern?) Why not write it so that an INPUT statement determines how many to add?
2. In the state of Connecticut, the sales tax rate is 7½%, which means that you must pay $0.08 on the first dollar, $0.15 on the first two dollars, $0.23 on the first three, and so on. Write a program that will print a Connecticut sales tax table on amounts from $1 to $100.

FOR-NEXT Loops Inside Each Other

Suppose you wanted to print a pattern of, say, six rows of asterisks (*). For example, you might want to begin a report with a six-row heading of asterisks. The following program works.

```
10 REM one way to get a heading of asterisks
20 FOR i = 1 TO 6
30 PR#1
40 PRINT "****************************************
****************************************"
50 NEXT i
60 PR#0
70 END
```

Now there's no problem with this program except it's not easy to keep count of eighty asterisks. Wouldn't it be easier to type on line 40:

```
print * (80 times)
```

This statement isn't allowed in SmartBASIC, but it is easy to imitate using a FOR-NEXT loop. If we use a semicolon, we get packed print (*packed print* is computerese for printing characters next to one another), so we can write:

```
FOR s = 1 TO 80
PRINT "*";
NEXT s
```

(There aren't any line numbers because we are going to replace line 40 with this block of lines.) Now put this into the program at line 40, either on one line using colons, or on lines 40–42. We end up with:

```
10 REM one FOR-NEXT inside another
20 FOR i = 1 TO 6
30 PR#1
40 FOR s = 1 TO 80
41 PRINT "*";
42 NEXT s
50 NEXT i
60 PR#0
70 END
```

This program would be easier to read if we could indent the inner loop. Unfortunately, this is not allowed in SmartBASIC. What we have just done is called *nesting* FOR-NEXT loops. Any time you want to change something systematically and then do something repeatedly, you will probably need two (or more) nested FOR-NEXT loops. The only rule to remember is that one loop must sit completely inside the other as in the following picture.

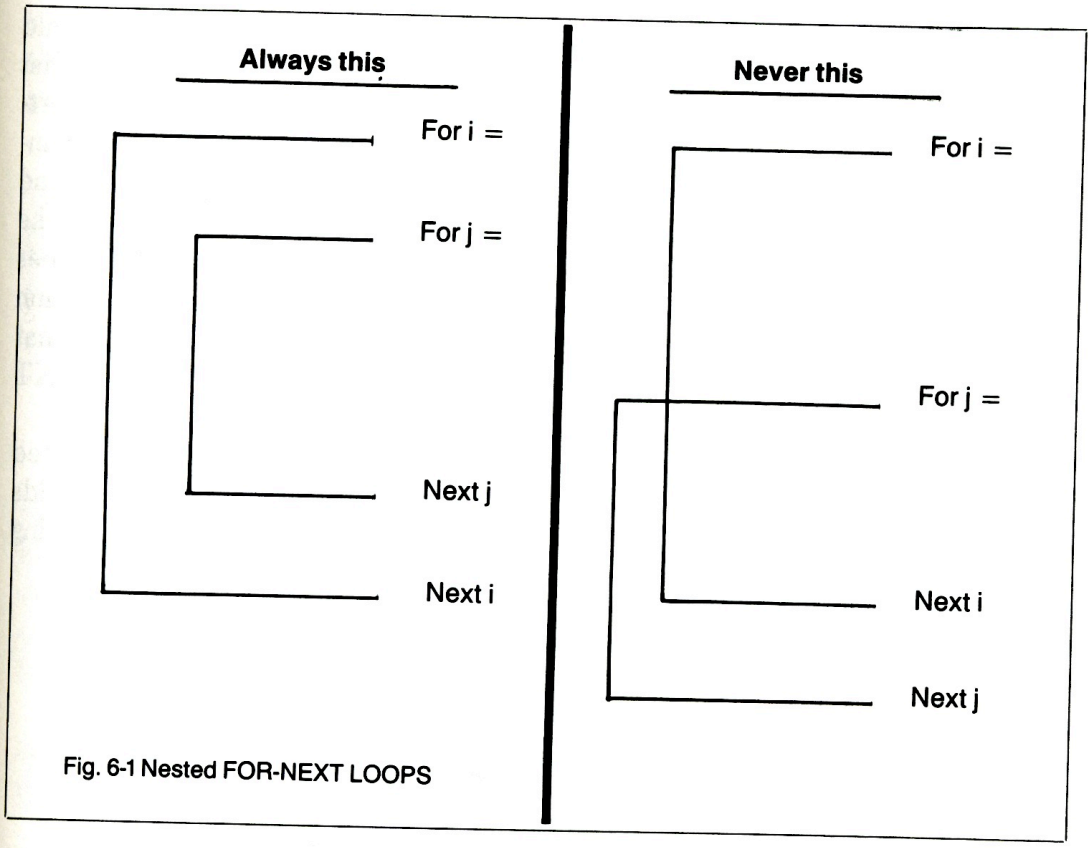


Fig. 6-1 Nested FOR-NEXT LOOPS

Suppose you have the following situation: you want to print out a multiplication table. For any one number it's pretty easy. Here's the program:

```
10 REM part of a multiplication table
20 INPUT n : REM n is the number for the table
30 FOR i = 1 TO 12
40 PRINT i;" * ";n; " =  "; i*n
50 NEXT i
```

How would you print out the whole multiplication table? You want *n* to have values from one to twelve and you want to change *n* using a FOR-NEXT loop. You might write a multiplication table program as follows:

```
10 REM   nested FOR-NEXT LOOPs
20 FOR j = 1 TO 12: REM the outer loop starts
30 FOR i = 1 TO 12: REM the inner loop starts
40 PRINT "    ";j;" * ";i;" = ";j*i
50 NEXT i : REM the inner loop ends
60 PRINT
70 NEXT j: REM the outer loop ends
80 END
```

This may look a little complicated the first time you see it. You should keep in mind that lines 30 to 50 do exactly one thing—they print out a list consisting of $j*1, j*2$, etc. So the inside loop is like a command in SmartBASIC which does something a bit more complicated than the usual commands. Once you think of lines 30 to 50 as a single operation, you will find the program easier to understand. We used some spaces in line 40 to make the table look prettier. The PRINT after we finished the inner loop does two things. It makes the output look nice (each part of the multiplication table is separated from the next) and it's a good way of seeing that each FOR-NEXT loop is finished before you go on to the next NEXT statement.

Here is another example. When we showed you timing loops, we pointed out that if the test value is very large, looping a thousand more times adds more than one second to the pause. A more accurate way to get a timing loop is to use the following bit of program.

```
                    .
                    .
                    .
50 INPUT "How many seconds do you want to pause? "; sec
60 FOR a = 1 TO sec
70 FOR p = 1 TO 1000
80 NEXT p
90 NEXT a
                    .
                    .
                    .
```

Think of lines 70 and 80 as saying "wait a second," and lines 60 and 90 as saying "again and again and again . . . . "

Nested FOR-NEXT loops have a bad reputation. They are supposed to be hard to program and even harder to understand. The reputation is undeserved. If you are careful about how you outline your program in English, you'll find them no more difficult than single FOR-NEXT loops.

We said there was one other way to slow ADAM down. The command is SPEED. SPEED controls how fast information is sent to the screen and to the printer. The normal speed for displaying things on the screen is quite fast; as we mentioned earlier, that's why you sometimes have to resort to CONTROL/S. In ADAM, the speed has a code number. The normal speed is

255, which is also the fastest speed. You can slow things down by making this number smaller—the slowest speed is 0. For example, the following program shows you the slowest setting.

```
10 REM slow speed
20 SPEED = 0: REM there is no LET in a speed command
30 FOR i = 1 TO 10
40 PRINT i
50 NEXT i
60 SPEED = 255
70 END
```

Run the program; the numbers from 1 to 10 crawl down your screen. Notice that we set the speed back to 255 on line 60. This is a good idea because otherwise ADAM will always use the last SPEED setting and it's very annoying to sit around waiting for the words to slowly appear on the screen. Try the following: remove line 60, run the program, and then (try to) list it. Do you see what we mean about annoying?

Here's a program that will demonstrate the speed settings and show you how to combine everything from this chapter. Why not take out a watch and time the settings?

```
10 REM a speed demo
20 FOR s = 0 TO 255 STEP 25
30 SPEED = s: REM starting at the slowest and going up by 25
40 PRINT "THIS IS SPEED "; s
50 REM begin the test print
60 FOR i = 1 TO 50
70 PRINT "    ";i
80 NEXT i
90 HOME: REM starting with a clear screen for the new speed
100 NEXT s
110 END
```

Self-Check 4

A. Questions

1. What is wrong with the following program?

```
10 LET SPEED = 0
20 PRINT "This is slow."
30 END
```

2. What is wrong with this one?

```
10 FOR i = 1 TO 20
20 FOR j = 2 TO n STEP .5
30 PRINT i*j
40 NEXT i
50 NEXT j
60 END
```

B. Programs

1. Write a program that will use a timing loop to test a person's mastery of the multiplication table.
2. Try to get the multiplication table to print out nicely on the printer. (This is called *formatting a print.*)
3. Try to arrange the multiplication table from the previous Self-Tests so that a heading is printed along the top and the left hand side.
4. Write a program that will print a list of the sum of the first ten, hundred, thousand, and ten thousand numbers. Can you see the pattern?
5. In order to save for retirement, a man deposits one thousand dollars when he is twenty-five, eleven hundred dollars when he is twenty-six, and so on. Assuming he gets 10% interest on his money, write a program that will figure out how much he has in the bank when he turns sixty-five.

Summary

In this chapter we showed you how to turn on the printer and turn it off. We showed you another way to program repetitive operations. You also learned the following commands:

FOR __ NEXT __ STEP __	forms a loop with a counter variable. The first value of the counter variable is the starting value. The step is added to the counter variable each time the NEXT statement is reached. The loop ends when the test value is passed.
PR#1	turns the printer on. Any further PRINT statements send their output to the screen and the printer.
PR#0	turns the printer off.
CONTROL/S	halts the execution of a program. The program restarts when any key is hit.
SPEED	controls how fast ADAM prints on the screen. The speed ranges from 0 (slowest) to 255 (fastest). The previous speed remains set until changed. When the SmartBASIC tape is loaded, ADAM starts at the fastest speed.

BUZZwords

BODY of a LOOP

COUNTER VARIABLE

DISABLE

FORMATTED PRINT

FOR-NEXT LOOP

HARD COPY

IMPLEMENTED

INITIALIZE

NESTED LOOPS

PACKED PRINT

STARTING and TEST VALUES for a counter variable

TIMING LOOP



Pictures

In this chapter, we'll give you a first look at how to use ADAM's color capabilities. You will learn the following commands in SmartBASIC:

GR	COLOR	VLIN
TEXT	PLOT	HLIN

as well as the color codes needed to draw pictures.

We've now shown you enough programming techniques to enable you to make effective use of ADAM's color capabilities. ADAM has two ways of handling color. In this chapter we'll show you *block* or *low-resolution graphics* (*resolution* means the amount of detail displayed). In chapters 14 and 15 we'll show you high-resolution graphics. (In computerese people say *high-res* or *low-res.*) The tradeoff is that in low-resolution graphics there are many more colors available. However, each block is fairly large (about one-sixth of an inch by one-third of an inch on a 15-inch television), so you are limited in what you can draw. In high-resolution graphics, there are fewer colors but each high-res block is so small that drawings can be quite intricate.

Getting Started

The command that gets you started is GR. GR stands for *graphics.* This is contrasted with *text. Text* handles words or symbols and *graphics* means pictures. Try using the command GR in direct (calculator) mode. The screen should blank out (if your television is well adjusted, it will be black) and the cursor should be about two-thirds of the way down. Anything you type or ask ADAM to print will appear in the remaining four lines. This is called the *text window.* You might want to type something and check this. The text window can hold four physical lines.

The command GR makes ADAM set aside a forty-by-forty *grid.* This means there are sixteen hundred blocks. Each one can be colored independently. Besides setting up the grid after the command GR, ADAM colors each of the blocks with the background color of black. This is why you can't see anything (yet). Look at figure 7.1. This shows you what your screen looks like to ADAM when it is in graphics mode. (We put numbers on the top and side—they are usually helpful.)

As you can see, the blocks are numbered from 0 to 39 on each side. For example:

Block 0, 0 is the left hand corner

Block 0, 39 is the right hand corner

Block 20, 20 is near the center of the screen

The first number gives the column and the second gives the row.

Until you enter the command TEXT, the screen will remain split between the large graphics area and the small text window. The command TEXT can be used either as a program statement or in direct mode. TEXT also erases whatever is on the screen.

Here is a short program that will give a sky-blue block in the center of the screen.

```
10 REM a sky-blue block
20 GR
30 COLOR = 14
40 PLOT 19,19
50 END
```

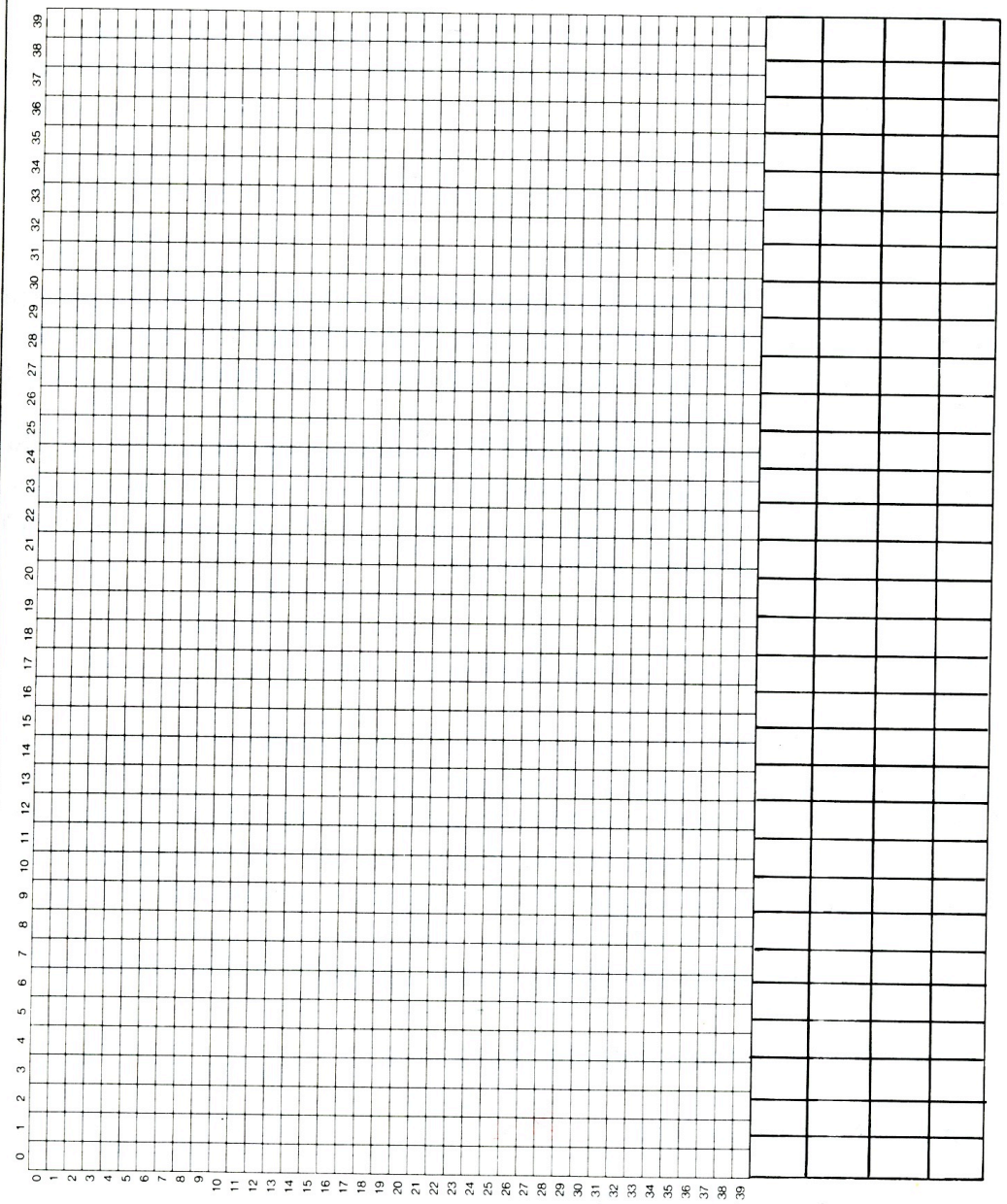

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

Fig. 7.1

The COLOR command tells ADAM the color it should use in coloring blocks. Each color has a number. For instance, 14 is sky blue. The PLOT command is the coloring command. ADAM fills in the block whose column is the first number (19) and whose row is the second number (also 19). To get a slightly bigger blue block we would add:

```
41 PLOT 20,19
42 PLOT 20,20
43 PLOT 19,20
```

If you add one or all of these commands and run the program you will see that moving to the left or right changes the first (column) number—smaller to the left, larger to the right. To move up make the second (row) number smaller. To move down make it larger.

We spent a lot of time on FOR-NEXT loops in chapter 6. They will be used frequently in this chapter. Using them, ADAM can draw pictures that are quite pleasing to the eye. Here's a program that will display a stair on a diagonal through the center of the screen.

```
10 REM a stair program
20 GR
30 COLOR = 14
40 FOR a = 0 TO 39
50 PLOT a,a
60 NEXT a
70 END
```

Let's go over this program.

Line 20 sets up the grid waiting to be painted.

Line 30 sets up the color—14 is sky blue.

Line 40 begins the loop.

Line 50 colors in the block whose column is the current value of *a* (0 in the first pass) and whose row is also the value of *a* (also 0 in the first pass).

Line 60 changes the value of *a* and sends us back for another pass through the loop.

Line 70 is reached after ADAM has finished the loop.

To draw your own pictures, you'll need a list of colors that ADAM has available in low-res graphics. These are listed, along with their code numbers, in table 7.1.

Code	Color	Code	Color
0	black	8	light yellow
1	magenta	9	medium red
2	dark blue	10	grey - 2
3	dark red	11	light red
4	dark green	12	light green
5	grey - 1	13	light yellow
6	medium green	14	cyan (sky blue)
7	light blue	15	white

Table 7.1 Colors in Low-Res Graphics

You should remember that the previous color stays on until the program changes it. (These colors might be slightly different on your set. On a black and white set, you'll get various shades of gray).

Although you can't see black (COLOR = 0) blocks, they are still important. When you replot a square with black, whatever was there disappears. You can use disappearing blocks to give the appearance of motion. This technique is used in video games and in animation (cartoons).

```
10 REM a screen walk
20 GR
30 FOR a = 0 TO 39
40 COLOR = 14
50 PLOT a,a
60 COLOR = 0
70 PLOT a,a
80 NEXT a
90 END
```

When you run this program, a blue block will run diagonally across your screen. If the block seems to move too fast, a timing loop (see chapter 6) can slow it down. For example, we used:

```
55 FOR p = 1 TO 200:  NEXT p
```

Experiment with different length timing loops until you find a delay that pleases you. What would happen if the timing loop was in a different part of the program?

Suppose you've drawn a color picture and would like to erase it. There are two ways: the command GR automatically repaints every block black. GR erases everything but still keeps you in graphics mode. The second way is to use the command TEXT. As we said before, this command takes you out of graphics mode, erases the screen, and puts the cursor in the left-hand corner. In fact, it's a more powerful version of the HOME command and can be used in the same way. The TEXT command makes the whole screen again available for displaying text.

Self-Check 1

A. Question

Where are the following points on ADAM's grid?

1. 39,39
2. 15,0
3. 27,39
4. 0,39

B. Programs

1. Write a program that will draw a dark green line 5 blocks wide across the center of your screen.
2. Write a program that draws a different screen walk. For example the block could move down the side or across the bottom.

More on Pictures

If you want to do anything complicated with graphics you should buy (or make) some graph paper. Make sure that the paper has enough squares to enable you to mimic ADAM's grid. You can also photocopy figure 7.2 on page 160. Using the right kind of graph paper makes it easy to see what your pictures will look like. The grid in figure 7.2 gives you a grid with blocks of the same shape as in ADAM's graphic display. It may be more useful than ordinary graph paper because it gives you a more accurate sketch. Don't be surprised if the picture on your screen doesn't seem exactly the same as your sketch. An optical illusion can affect what you see when something is displayed on a screen. For example, a perfect circle might look lopsided.

Here's an example of using a grid to lay out a picture. We wanted to have the word ADAM appear in large block letters near the center of the screen. We took a copy of figure 7.2 and then we blocked out each of the letters as in figure 7.3.

Now it's easy to write a program to print each letter. Here's one for the letter *A*:

```
10 REM program to draw the letter A
20 GR: COLOR = 14
40 FOR j = 19 TO 22
50 PLOT 8,j: PLOT 12,j
60 NEXT j
70 PLOT 9,20: PLOT 10,20: PLOT 11,20
80 PLOT 9,18: PLOT 11,18: PLOT 10,17
90 END
```

(You can try to program the other letters. You have to add lines to the program above to color the blocks laid out in figure 7.3.)

Typing lots of PLOT statements quickly gets tedious and programming loops to paint horizontal and vertical strips is a bore. For this reason, SmartBASIC has two commands that simplify drawing horizontal and vertical stripes. They are VLIN and HLIN. As the names suggest, VLIN draws vertical lines, and HLIN draws horizontal lines. You have to tell ADAM where to put a line, where it should start, and where it should end.

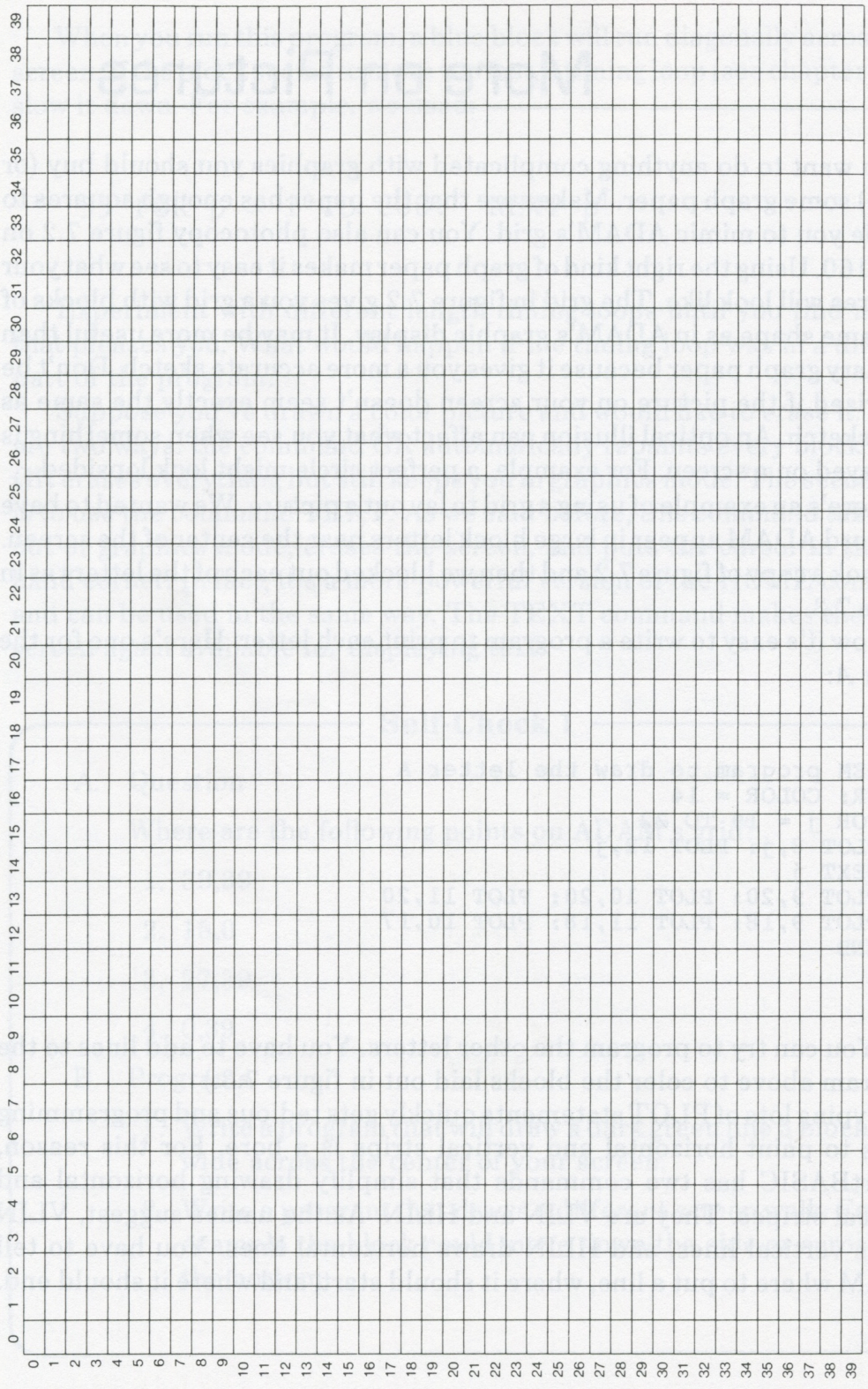

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

Fig. 7.2

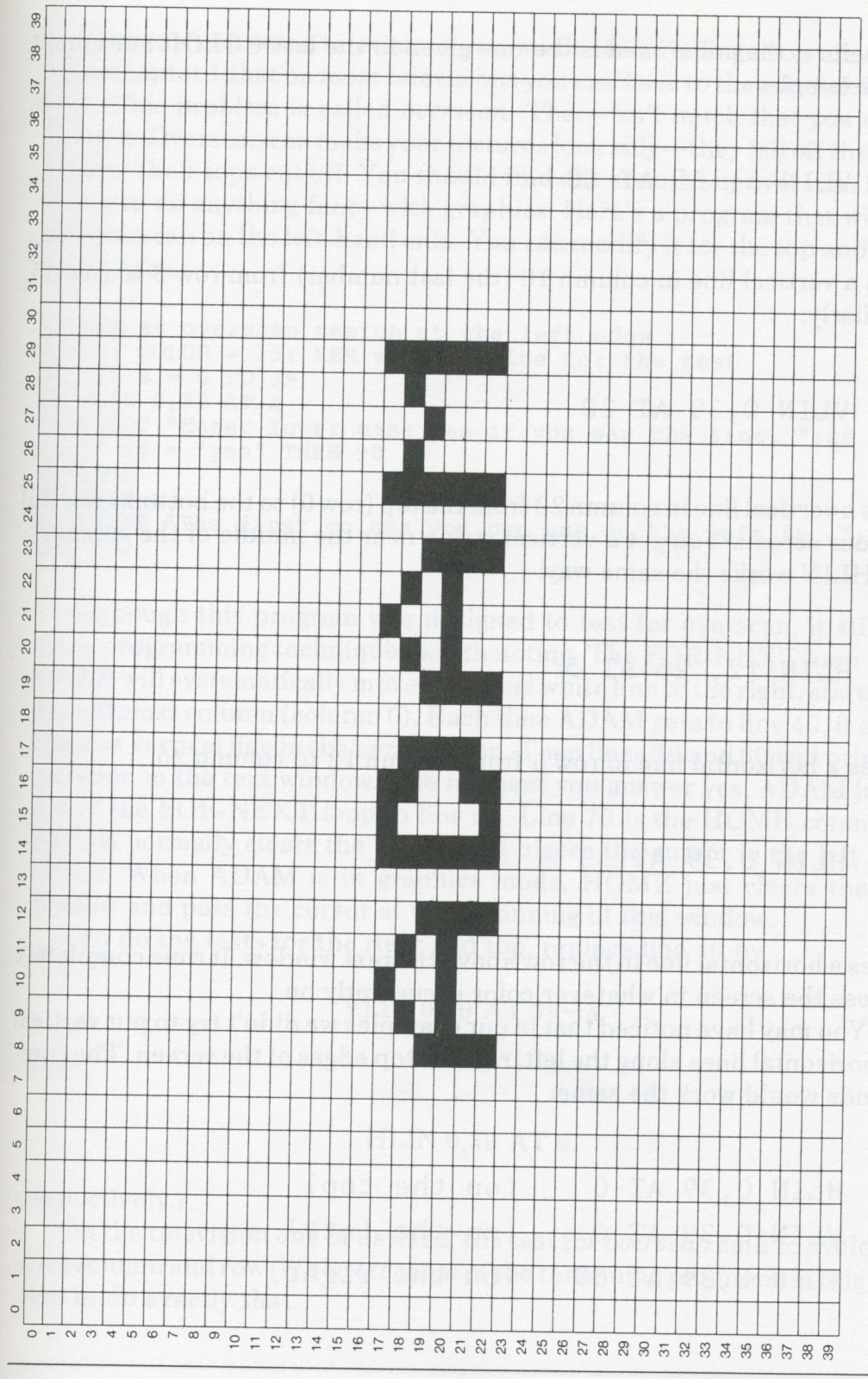


Fig. 7.3

As before, the color used is the one given in the last COLOR statement. For example:

```
VLIN 5, 35 AT 10
```

puts a vertical line in column 10 (the last number) from row 5 to row 35. Similarly:

```
VLIN 0,39 AT 20
```

puts a vertical line in column 20 from the top (row 0) to the bottom (row 39) of your screen. You get a vertical stripe near the middle of the screen.

HLIN works the same way:

```
HLIN 12, 25 AT 5
```

gives a horizontal line in row 5 from column 12 to column 25.

```
HLIN 0,39 AT 39
```

gives a horizontal line in the row above the text window. It runs completely across the screen in whatever color is currently on.

You may have noticed that in our examples we didn't try to put vertical or horizontal lines along the left, right or top edges of the screen. The commands would work the same:

```
HLIN 0,39 AT 0     (on the top)
VLIN 0,39 AT 0     (on the left)
VLIN 0,39 AT 39    (on the right)
```

However, on some televisions you won't see anything. You may recall from chapter 1 that on most televisions you can't see to the edge of the picture. This problem is called *overscan.* There isn't much that you can do about it. Overscan can make your pictures look silly—they fall off the edges or have their tops cut off. You should find out about your overscan before trying to do anything fancy with graphics. Here's a program that will find the overscan on the left-hand side. You can modify it for the top and right-hand sides.

```
10 REM an overscan search at the left edge
20 GR: COLOR = 15: REM we use white for the test
30 FOR a = 0 TO 39
40 VLIN 0,39 AT a
50 INPUT "Enter lower case yes if you saw the line. ";s$
60 IF s$ = "yes" THEN 90
70 HOME
80 NEXT a
90 PRINT "The first column you can use on the left is  ";a
100 END
```

Although this program was designed to test for overscan, it still has some programming techniques worth noting. The FOR-NEXT loop (lines 30–80) will systematically move a vertical white line to the right, starting in the leftmost column (column 0). Each time ADAM gets to line 40, it draws another vertical line in the next column. Then lines 50 and 60 will print the question in the text window. The moment you answer yes, ADAM jumps out of the FOR-NEXT loop to line 90. Line 70 is the HOME command. HOME normally clears the screen and places the cursor in the left hand corner. When ADAM is in graphics mode, HOME just clears the text window and puts the cursor at the beginning of this window.

(To do the tests for the right and top, replace line 40 by

VLIN 0,39 AT 39−a,

and

HLIN 0,39 AT a,

respectively.)

On the television one of us used, the test for *overscan* said to avoid the first column and row. We wrote most of the remaining programs in this section to do exactly that.

Now that we have shown you the VLIN and HLIN commands, it is easy to write a program to display the colors. Here's an outline of how to fill the screen with magenta (COLOR = 1):

1. Set COLOR = 1.
2. Put a horizontal line on the top across the whole screen.
3. Put a horizontal line across the next row.
4. Continue until all 40 rows are filled in.

Here's the program:

```
10 REM a magenta wash
20 GR
30 PRINT "This is color # 1"
40 COLOR = 1
50 FOR a = 0 TO 39
60 HLIN 0,39 AT a
70 NEXT a
80 END
```

Now let's do this for all the colors by putting this program inside another FOR-NEXT loop. The COLOR has to go from 0 (black) to 15 (white), and we want to erase the previous color code (line 30) each time.

Here's the program:

```
10 REM an annotated color display
20 GR
30 FOR c = 0 TO 15
40 PRINT "This is color # ";c
50 COLOR = c
60 FOR a = 0 TO 39
70 HLIN 0, 39 AT a
80 NEXT a
90 HOME : REM erase the number of the previous color
100 NEXT c
110 END
```

This program paints the new color over the old one row by row. We think that the effect is attractive. Do you see how to erase the screen before starting a new color?

Using the color list given in table 7.1 and this program, you can compare the colors that your television displays with the colors that ADAM sends to it. Many sets will let you adjust your colors until they match ADAM's.

It's possible to make quite fancy displays by using variables in the VLIN and HLIN commands. Let's write a program that will draw colored frames one inside the other. The display is called a "nest of frames." Because of the overscan problem, we'll start in the first row and column. A frame is just four lines that meet. By laying out the frames on a piece of graph paper you can decide which commands are necessary to draw a single frame. For example, the following commands draw the largest frame:

```
HLIN 1,38 AT 1     (top)
VLIN 1,38 AT 38    (right side)
HLIN 1,38 AT 38    (bottom)
VLIN 1,38 AT 1     (left side)
```

Now suppose you wanted to make the next smaller frame. For example, we could move toward the center by two rows and two columns. The commands would be:

```
HLIN 3,36 AT 3
VLIN 3,36 AT 36
HLIN 3,36 AT 36
VLIN 3,36 AT 3
```

It's a good idea to experiment with these kinds of commands until you are comfortable with drawing various kinds of frames.

We are going to vary the colors and start in row 1 and column 1. Each time the color changes we want the frame to be one size smaller. Here's the program:

```
10 REM a nest of frames
20 GR
30 FOR c = 1 TO 15
40 COLOR = c
50 HLIN c,39-c AT c
60 VLIN c,39-c AT 39-c
70 HLIN c,39-c AT 39-c
80 VLIN c,39-c AT c
90 NEXT
100 END
```

The last program we want to show you in this section will display as successive vertical bars all the colors ADAM has available. Since there are sixteen colors and forty columns available to draw them in, we'll make each bar two columns wide.

```
10 REM ADAM's spectrum
20 LET m = 4: REM start in the fourth column
30 GR
40 FOR c = 0 TO 15
50 COLOR = c
60 VLIN 10,30 AT 2*c+4
70 VLIN 10,30 AT 2*c+5
80 NEXT c
90 END
```

Notice that the counter variable *c* is used both to set the color and to tell ADAM which vertical lines to paint with that color. When $c = 0$, ADAM paints columns 4 and 5; when $c = 1$, ADAM paints columns 6 and 7; and so forth. In lines 60 and 70, ADAM will do the calculation before deciding which column to paint. Instead of using a constant or the name of a variable, a calculation leads to a numeric value (number). You can do computations in many SmartBASIC commands. We already did this with the commands PRINT, LET, and FOR _____ TO _____ STEP _____. 2*c+4 is an example of a *numeric expression.* This buzzword means something that ADAM can eventually turn into a number.

Self-Check 2

A. Questions

1. What does the command VLIN 3,33 AT 20 do?
2. What does the command HLIN 1,39 AT 39 do?

B. Programs

1. Have ADAM draw a green line across the bottom of the graphics area of your screen.
2. Use VLIN and HLIN to program the other letters in the name ADAM.
3. Draw a blue line in every other row.
4. Modify the program you just wrote so that the blue line alternates with a red line.
5. Draw an *L* shape in red.
6. Draw a large *Z* in three different colors.

A Bar Graph Program

In this section we are going to show you the first of two longer programs. The first program makes bar graphs. You might want to just try using this program and read the description later.

A bar graph is a picture showing a comparison between things by piling up blocks. The piles show the relative sizes of the things. For example, if you have seventy-five books and your friend has twenty-five, you could compare these numbers by making piles of blocks. Your pile would have three blocks and your friend's would have one. ADAM can show these comparisons by making the piles from low-res graphics blocks. The comparison is very attractive when the piles have different colors. Bar graphs are usually labelled across the bottom and have a scale along the left edge showing the size of the piles. The left edge and the bottom are called the *axes.* The horizontal edge is usually called the *x-axis* and the vertical edge is the *y-axis.*

In our bar graph, the colored bars (and labels) will stand for three different months. We'll also have the x- and y-axes in white along the bottom and left edges, respectively.

The only thing that is tricky about a program to draw bar graphs is that the bars start from the bottom of the screen and go up. Each bar represents

some number of things that happened in that month. You could have seventeen units in the first month (a unit could be the sale of a car or millions in sales for a large company or even the number of times you went shopping). The bar starts at row 38, so a bar up to row 38 represents one unit. The bar has to go up sixteen more units until it ends at row 22. Since ADAM has 39 rows—remember, we're not using the top row because of overscan problems—we will allow up to thirty-eight units. If you had, say, seventy-six units in a month, then each row would have to stand for two units. It wouldn't be hard to have ADAM calculate a scale based on the information you gave it. Since this program is going to be fairly long as it is, we won't do scaling here.

Here is an outline of the program:

1. Get the data for the various months.
2. Display the axes.
3. Choose a color for the first month and display the bar.
4. Move over a few columns and display the second bar.
5. Do the same for the third month.
6. Print a title.

Here is the program:

```
10 REM a bar graph program
20 HOME
30 PRINT "This program will display a bar graph for 3 months"
40 PRINT "You must make up the units yourself."
50 PRINT "You can use up to 38 units in any month."
95 REM
96 REM
97 REM ***********************************
98 REM data gathering module
99 REM ***********************************
100 PRINT "What are the numbers for the 3 months?"
110 PRINT "Enter the 3 amounts separated by commas."
120 INPUT m1,m2,m3
130 IF m1<39 AND m2<39 AND m3<39 THEN GOTO 200
140 PRINT "Your data is out of range."
150 FOR i = 1 TO 5000: NEXT i
170 GOTO 430
195 REM
196 REM
```

(continued)

```
197 REM ***********************************
198 REM graphing module
199 REM ***********************************
200 GR
210 COLOR = 15: VLIN 0,39 AT 0: HLIN 1,39 AT 39
220 FOR s = 14 TO 17
230 IF m1 < 1 GOTO 250
240 COLOR = 2: VLIN 39-m1,38 AT s
250 IF m2 < 1 GOTO 270
260 COLOR = 3: VLIN 39-m2,38 AT s+6
270 IF m3 < 1 GOTO 290
280 COLOR = 4: VLIN 39-m3,38 AT s+12
290 NEXT s
300 PRINT "MONTH =      1     2     3"
395 REM
396 REM
397 REM ***********************************
398 REM run again module
399 REM ***********************************
400 PRINT "Do you want to draw another graph?"
410 INPUT "(Enter y or n) ";yn$
420 IF yn$ = "y" OR yn$ = "Y" THEN TEXT: GOTO 100
430 TEXT
440 END
```

This program is much longer than any previous one in this book. As with any long program, it was written in modules. (Recall that *module* is the buzzword for a piece of a program.) We used a different format for displaying (and typing) the modules. An example is the module between lines 95 and 96. Lines 95 and 96 are blank. The only reason for putting in the command REM is that completely blank lines are never recorded in ADAM's memory. We want blank lines to mark the boundary between modules. Lines 97 and 99 are a frame for the title of the module. The *data-gathering module* is similar to many previous programs or fragments. This is also true for the *run again module.* We used a string variable named *yn$* to hold the response to the yes or no question.

Most of the modules are similar to programs we've already shown you. This will happen more and more. After you first decide what you want to do, you'll find that writing a program often consists of patching together modules you've already seen (perhaps in a different context). You make some small changes, write a control module, and you are done.

The IF-clause on line 130 has three separate tests. They check whether the data is out of range. When the IF clause is false, the data is out of range, so the PRINT statement tells you so and the program ends by going to line 430. Between lines 140 and 170 there is a timing loop so that you can read the statement telling you that your data is out of range. You should also

notice that the GOTO command on line 170 doesn't send ADAM to the END. ADAM is sent to the TEXT command before going to END. It is a good idea to leave a graphics program by first returning to TEXT. (You should make sure that the program doesn't erase the drawing before you get a chance to look at it. One good way to create a pause is to have the program ask you a question, using an INPUT command, before the TEXT command.)

The only new ideas in this program are contained in the "graphing module." Although it is only eight lines long, it does many things. First, line 210 plots the axes for the graph. All the bars are four columns wide and are plotted in four passes through the loop. Each pass adds one column to each bar. Before ADAM tries to plot a bar, it checks to see that the size of the bar is in the allowable range. Then the color is set and ADAM starts plotting.

You might be wondering what would happen if you entered 2.2 units for the first month. Line 240 would tell ADAM to plot a bar between rows 38 and 36.8. Of course there isn't any row 36.8 so ADAM would truncate the .8 and it would plot the bar up to the 36th row.

Warning: If you ask ADAM to plot anything but a whole number it will truncate the decimal part. Since it doesn't round off numbers, your graphs (or pictures) can become inaccurate. If you ask ADAM to do anything outside its range of 0 to 39, you will get an error message and the program will bomb.

Self-Check 3

A. Questions

1. In the bar graph program, suppose the values you want to plot are 78, 46, and 23. What does each unit have to represent?
2. Repeat question 1 with values 159, 49, and 83.
3. Do you see how the largest value and 39 are related when you try to scale the data?
4. What would happen if we didn't test the values of *m1*, *m2*, and *m3* in the graphing module?

B. Programs

1. Write a bar graph program for a four-month period.

2. Sometimes we make bar graphs to show the difference between our income and our expenses. Both are plotted for each month in the same bar but the difference is shown in red if expenses are more than income and in blue in the other case. Write a program to compare income and expenses for three months with the difference shown in red or blue. (Note: This is harder to program than an ordinary bar graph. You have to compare the income and expenses and choose colors based on the comparison.)

The Low-Res Graphics Editor

This section contains our "paintbrush" program, or low-res graphics editor. This program is even longer than the bar graph program. Because it is so long, we had to carefully outline the program and build it in pieces. To give you an idea of how to write a long program, we'll take you through the steps we used. If you don't want to work through the details of such a long program but still want to use it, we suggest entering it and then saving it. Later you might want to see how it works and how it was written.

SOME WORDS OF CAUTION.
WE SUGGEST BUYING A FEW EXTRA DIGITAL DATA PACKS. USE THEM TO MAKE BACKUPS OF ANY PROGRAMS THAT YOU WOULDN'T LIKE TO HAVE DESTROYED. [*BACKUPS* ARE COPIES OF PROGRAMS.] DIGITAL DATA PACKS ARE RELIABLE BUT THEY WILL EVENTUALLY WEAR OUT; THEY CAN ALSO BECOME USELESS BECAUSE OF DUST OR CARELESS HANDLING. REWRITING OR REENTERING A LONG PROGRAM IS ENOUGH WORK THAT IT'S WORTHWHILE TO KEEP COPIES.

The first step in writing a program is to decide what you want it to do. We wanted to write a program that lets you move a paintbrush around ADAM's grid and then use that brush to paint any block that you want. We wanted to permit the use of any of the sixteen colors that are available in low-res graphics. Suppose the paintbrush is at a particular block. You should be able to move it up or down, left or right. The program should also be able to show the present position of the paintbrush as well as let you stop the program gracefully (without using CONTROL/C).

Then we started outlining the program. The painter—the person controlling the paintbrush—should be presented with a choice of things to do:

1. Find out where the paintbrush is.

2. Stop painting.

3. Move in any of four directions.

4. Decide to paint some color at the present position of the paintbrush.

5. After taking one of the above actions, the painter should be able to take further actions.

This was a good first outline but it wasn't enough. For example, suppose the painter moved the brush off the screen and asked ADAM to paint something off the sides. This would cause an error message; the program would automatically end and everything that had been painted would be lost. So before doing step 4 (or step 1) we should check that the new location is allowed. (In any complicated program, you try to *trap* or test for illegal commands before they bomb your program. We did this in the bar graph program.) It is rarely possible to trap every problem that might arise; usually programmers are satisfied to trap the most common problems.

Before we started writing the program we thought about how ADAM would keep track of the position of the paintbrush. To do this we need two variables—one for the column and one for the row. Naturally, we decided to call them *col* (for column) and *row*. Each time the painter asks to move left, ADAM should subtract one from the variable named *col*; each time the paintbrush moves down ADAM should add one to the variable named *row*; and so on. The initial values for *col* and *row* determine where the paint-

brush starts. To start in the center of the grid, ADAM must start with *col* = 20 and *row* = 20. (In computerese, we say that we *initialize* both of the variables to have value 20.)

Next we wrote an outline that showed the different modules that we would need. In long programs the first module always has the same form. It tells the operator what program is running, gives some directions, and asks whether to proceed or end the program. This is usually a good idea because you can run a program by mistake and it may take two hours to run. Of course you can always stop any program by using CONTROL/C, but sometimes you will destroy information on the data pack by stopping a program in the middle (see chapter 16).

In a program like the "paintbrush," where there are many choices, the second module should be a menu. Since moving the paintbrush is a simple command, we decided to include the moves inside the menu. The menu or "choice module" could also contain the exit from the program (the END statement). It is a good idea to keep instructions on the screen so that the operator is told what he or she can do. We thought that these instructions could be part of the "choice module" and just stay on the screen while other modules were being processed.

Other modules that you sometimes find in long programs are error-trapping modules and exit modules. Exit modules are particularly important when you use files (again see chapter 16).

From our original outline, we knew we needed one module for actually painting the colors on the screen and another for locating the position of the paintbrush. It turned out that the "locate module" was the most interesting (and hardest) to write. Here's how we did it.

There are several ways that you can show the painter the position of the brush. The easiest to program is by printing the row and column at which it is located. This can be done with one line:

```
PRINT "the brush is positioned at
col = ";col;" and row = "; row
```

Unfortunately, we usually couldn't see where the paintbrush was located using only these numbers. We decided to catch the painter's attention by having the position of the brush flash on the screen. (When a block flashes it

changes color.) If the block had already been colored, we wanted to have ADAM return the block to its original color after flashing. ADAM will have to remember the original color of the block to do this. As soon as we knew exactly what we wanted the module to do, we could write a detailed job description for the module.

1. Check that the current *row* and *col* are allowable positions (that they lie on the graphics screen).

2. Find out what color that position is currently painted.

3. Paint over that position with black.

4. Alternately paint the position black and white with enough time delay so that it can be seen.

5. Go back to the original color.

Step 1 is an IF-THEN. Steps 3, 4, and 5 use COLOR and PLOT commands with timing loops. We've done all those things before. Step 2 is the problem. So far we haven't introduced a command that can examine the screen and tell us the color that is painted at a particular location in the grid. We knew what we wanted to do but didn't know how to do it. Whenever this happens on ADAM or any other computer, we start going through a reference manual that lists all the available commands (for example, the reference section of the SmartBASIC manual or the *Ready Reference Guide* included with this book.) If we're lucky, we find exactly the command we need; if we're not, we must either program around the problem or rethink the whole module (or even the whole program). SmartBASIC has the command we needed.

The command we needed is SCRN (*exprnm*$_1$, *exprnm*$_2$). *exprnm*$_1$ is the column and *exprnm*$_2$ is the row. SCRN(col,row) gives the code for the color in the block at the current values of *col* and *row*. Once we had this command we were able to translate our outline into another outline. The second outline was very close to the lines that we needed in the module. Because it is already close to being in SmartBASIC statements, we say that this outline is in *pseudo-code.* This buzzword means statements written some where between standard English and a programming language. We aren't sure where English stops and pseudo-code begins. Everyone seems to have their own definition of pseudo-code. An example in ours is:

if col < 0 or col > 39 then problems

if row < 0 or row > 39 then problems

c = scrn(col,row)

color = 0: plot color: delay loop

color = 15: plot color: delay loop

loop the previous two steps (for flashing)

color = c: plot color

go back to the menu

You should notice that at this stage we didn't say what we wanted to do if we had problems. ADAM had to be sent to some module that does error trapping, but we hadn't written that module yet.

We next made the following list of modules with blocks of lines set aside for each module (we left plenty of extra lines in case any module turned out to be much longer than expected). In general, no module should be longer than about one page (25 lines). If a module gets too long, you should try to break it up into smaller modules.

LIST OF MODULES

1. Direction module (lines 10–490)
2. Choice module (lines 500–990)
3. Locate module (lines 1000–1490)
4. Painting module (lines 1500–1990)
5. Error trapping module (lines 2000—)

Table 7.2

After we laid out each module, we started writing the program. Then it was easy. For example, to write the *locate module* we had to use line

numbers, problems became a GOTO statement, the delay loop became a timing loop and we wrote a loop around lines 4 and 5.

Before you work through the program you might try laying out another module using the same tricks that we used. Now to the program.

```
 10 REM  the paintbrush
 20 HOME
 30 PRINT "this is a paintbrush program"
 40 PRINT "the moves are:  u=up, d=down"
 50 PRINT "the moves are: l=left, r=right"
 60 PRINT "use p to paint"
 70 PRINT "use w to find out where you  are"
 80 PRINT "use e to exit"
 90 INPUT "enter s to start"; s$
100 IF s$ = "s" THEN 500
110 PRINT "You entered something strange."
120 PRINT "Type run to start again."
130 FOR pause = 1 to 2000: NEXT
140 TEXT
150 END
495 REM
496 REM
497 REM **********************************
498 REM   choice module
499 REM **********************************
500 GR
510 row = 20: col = 20
520 HOME
530 PRINT "u=up, d=down, l=left, r=right"
540 PRINT "w=where, p=paint, e=exit"
550 INPUT ch$
560 IF ch$ = "u" THEN row = row-1: GOTO 520
570 IF ch$ = "d" THEN row = row+1: GOTO 520
580 IF ch$ = "l" THEN col = col-1: GOTO 520
590 IF ch$ = "r" THEN col = col+1: GOTO 520
600 IF ch$ = "e" THEN 150
610 IF ch$ = "w" THEN 1000
620 IF ch$ = "p" THEN 1500
630 HOME: PRINT "TYPO. RETRY"
640 FOR p = 1 TO 2000: NEXT p
650 GOTO 520
995 REM
996 REM
997 REM **********************************
998 REM   locate module
999 REM **********************************
1000 IF col < 0 OR col > 39 THEN 2000
1010 IF row < 0 OR row > 39 THEN 2000
1020 c = SCRN(col, row)
1030 FOR i = 1 TO 8
1040 COLOR = 0: PLOT col, row
```

(continued)

```
1050 FOR j = 1 TO 250: NEXT j
1060 COLOR = 15: PLOT col, row
1070 FOR j = 1 TO 250: NEXT j
1080 NEXT i
1090 COLOR = c: PLOT col, row
1100 GOTO 520
1495 REM
1496 REM
1497 REM ********************************
1498 REM painting module
1499 REM ********************************
1500 IF col < 0 OR col > 39 THEN 2000
1510 IF row < 0 OR row > 39 THEN 2000
1520 INPUT "What color do you want?"; c
1530 COLOR = c: PLOT col, row
1540 GOTO 520
1995 REM
1996 REM
1997 REM ********************************
1998 REM  off-screen trapping module
1999 REM ********************************
2000 HOME: PRINT "You have moved off-screen."
2010 PRINT "The brush has been moved back to the center."
2020 PRINT "Try again."
2030 GOTO 510
```

We'll go over each module separately.

The entry module for the program uses lines 10 to 150. The program starts and ends there, but it really doesn't do anything. It clears the screen and gives some instructions on how to use the program. When ADAM sees the INPUT command it stops and waits for you to enter something at the keyboard. If you are ready to run the program, you enter *s*. Hitting any other key immediately ends the program. You'll notice that we left several extra spaces on line 70. This makes the screen display look better.

After starting, ADAM goes to the choice module in lines 500 and 650. First, this module puts ADAM in graphics mode and prints directions in the text window. The paintbrush starts in the center of the grid. By following the directions, you can choose what the paintbrush should do. If it is to be moved, the move is made by changing the values of the *col* and *row* variables. Then ADAM goes back to line 520. The directions are reprinted in the text window and the paintbrush awaits further orders. The reprinting is done almost instantaneously. As fast as you can choose a direction and hit ENTER, the paintbrush gets ready for its next action.

In line 600 you can choose to stop painting. (This INPUT statement also allows you to look at what you've painted so far.) If you choose to have the brush paint the block where it is located, the program goes to the paint

module. If you want to find out where the paintbrush is, ADAM is sent to the locate module.

The paint module first checks to see whether the brush is positioned on the grid. Asking ADAM to PLOT outside the grid bombs the program unless it traps the illegal command. If the brush is off the grid, ADAM sends it back to the center without doing any painting (by going to the off-screen trapping module). If the block is on the grid, the program asks for a color and paints it using that color. It then goes back to ask for more instructions.

We enjoyed writing this program because of the different tricks we had to use to make it work well. Even if you don't like writing very long programs, you can enter the paintbrush program and use it. We think it is fun to use.

Self-Check 4

A. Questions

1. What do you think would happen if you typed in a = SCRN(3,55)? Try to do this using different values inside the parentheses. What can you conclude about the allowable values inside the parentheses?
2. In the paintbrush program, how could you have ADAM type the column and row position of the paintbrush?
3. In the paintbrush program, what happens when you don't make a choice of one of the possible actions?
4. If you paint a picture using the paintbrush and then exit from the program, what mode are you in?

Summary

In this chapter, you started using ADAM's color capabilities. We showed you how to use the following commands:

GR turns on ADAM's low-resolution color graphics. This sets up a forty-by-forty grid, numbered 0 to 39 in each direction. You can color any block by using the PLOT command.

COLOR selects a color code (see table 7.1) for ADAM to use in painting blocks.

PLOT is the command that actually paints a block in the previously chosen color.

VLIN draws a vertical line.

HLIN draws a horizontal line.

TEXT takes ADAM out of its graphics mode, clears the screen and moves the cursor to the upper left corner of the screen.

We also told you how important it is to back up your long programs by saving them on more than one digital data pack.

BUZZwords

BACKUP
GRAPHICS
HIGH RESOLUTION
LOW RESOLUTION
NUMERIC EXPRESSION
OVERSCAN
PSEUDO-CODE
TEXT
TEXT WINDOW
TRAP



Typewriter Art and Pretty Printing

In the last chapter, we showed you how to draw pictures on the screen in low-resolution graphics. In this chapter, we'll show you how to use the SmartWriter printer to make hard copy pictures.

You will learn the following commands:

INVERSE	FLASH	NORMAL	SPC
VTAB	HTAB	CHR$	TAB

We will also show you how to use the control codes to gain full power over ADAM's printer and screen.

Typewriter Art

While it is true that ADAM's SmartWriter can't print in color, nonetheless you can make quite pleasing displays with it. Typewriter art is done in much the same way as low-resolution graphics, except ADAM prints characters on paper instead of colored blocks on the screen.

The SmartWriter printer can print eighty characters on each of sixty-five lines. If you think of each of these characters being placed in a block, then the printer sees an ordinary piece of paper as a grid with eighty blocks on each of sixty-five rows (see figure 8.1). To create a typewriter art drawing, blacken the blocks on the grid until you are satisfied with the design. Then you tell ADAM to print an *X*, a *, or some other symbol at the positions corresponding to the pattern on the grid.

You may have seen posters of "printer art." (There are posters ranging from the Mona Lisa to Snoopy.) To get really intricate designs you need delicate overwriting to control shading. In this chapter we won't get very involved with overwriting—it's possible to do it with ADAM but usually it's too complicated. (Overwriting is done by backspacing or printing backwards.)

Suppose you want to put an *X* in various boxes. In theory you could do this by just giving the command PR#1 and being *very* careful about your PRINT commands. You would have to count out dozens of spaces inside quotation marks and would quickly give up. ADAM has two commands that make inserting spaces easier: SPC and TAB. Both work only in PRINT statements and work either on the screen or with your SmartWriter.

The cursor position shows where the next character will appear on the video display. On the screen the command SPC inserts a specified number of blanks by moving the cursor. It does something similar with the printer. Instead of a cursor, the SmartWriter has a *printhead.* (This buzzword means the part of the SmartWriter that actually prints.) SPC moves the printhead and acts just like the space bar on a typewriter. For example:

```
PRINT "HELLO"; SPC(15);"GOODBYE"
```

prints the word *goodbye* fifteen spaces after the word *hello.* If the SmartWriter printer were turned on (by the PR#1 command) then this would happen both on the screen and on the printer.

Fig. 8.1 **(GRIDS IN THIS CHAPTER NOT PROPORTIONAL TO STANDARD PAGE SIZE)**

A numeric expression (recall that this is the buzzword for a number, variable, or calculation) can also be inside the parentheses. The value of the numeric expression must be a number between 0 and 255. If the value is a decimal, it will be truncated. On your screen there are thirty-one spaces per line; on the printer, there are eighty.

TAB is the other command. TAB works almost the same way as SPC except TAB always starts counting from the first position on the logical line. This means TAB(10) is always the tenth position in from the left. Unfortunately, you can't use TAB to backspace. If the cursor (or the printhead) is past the tenth position this command does nothing. (It seems that, in the earlier versions of SmartBASIC, the TAB command worked strangely for values above thirty-one. For example, TAB(32) didn't move the printhead to the thirty-second column. In the later versions of SmartBASIC this problem is supposed to be corrected. If you use one of those data packs, TAB is a good command for screen displays, but not reliable for printed ones.)

To illustrate the use of the TAB command, here is a program that draws a ramp using only the letter *X*.

```
10 REM  a ramp
20 PR #1
30 FOR i = 1 TO 75
40 PRINT TAB(i); "XXX"
50 NEXT i
60 PR #0
70 END
```

Line 20 turns the printer on so that all PRINT statements go both to the screen and the printer.

The FOR-NEXT loop starts in line 30. In the first pass through the loop, ADAM starts printing in column 1, in the second pass ADAM starts in column 2, etc. In the last pass ADAM starts in the seventy-fifth column. Sometimes programmers say that "in the *i*th pass we're in the *i*th column." This sounds like a lot of algebra, but when people start speaking this way, say (silently) to yourself, "i" is a number so they mean: in the first pass we're in the first column, in the second pass the second, and so on. Using the word "*i*th" here is really shorthand for the "and so on."

Notice that line 60 turns off the printer. As we mentioned before, it is

always a good idea to shut off the printer before the program ends.

This program is formatted (arranged) so that the hardcopy is a picture of a ramp. On the screen, the display moves back to the first column after printing in the thirty-first column. The screen display of typewriter art often appears very different from the printed copy.

Using the TAB and SPC commands, it's not hard to draw a simple picture in a short program. If the object is symmetrical, we can use these commands in loops. Otherwise the programs become long lists of complicated PRINT statements.

To show you what we mean by a symmetrical object, the program below draws a large 'X' on a piece of paper. Look at the grid in figure 8.2 on next page.

We drew it so that there is space on each side of the *X*. This makes it appear more symmetrical. The program is a bit tricky; we suggest thinking about it before reading our explanation.

```
10 REM an X
20 PR#1
30 FOR i = 0 TO 23
40 PRINT SPC(10 + i);"XXX";SPC(46 - 2*i);"XXX"
50 NEXT i
60 REM now the bottom part
70 FOR j = 23 TO 0 STEP -1
80 PRINT SPC(10 + j);"XXX";SPC(46 - 2*j);"XXX"
90 NEXT j
100 PR#0
110 END
```

Let's look at the crucial line 40 more closely. In the first pass $i = 0$. This means that ADAM moves the printhead ten units, prints XXX, then moves it forty-six more units and prints XXX again. In the second pass i = *1*. So now ADAM spaces eleven before the XXX, *but* only 44 (46 — 2*1) before the second XXX. This is what we need to have the *X* narrow by two columns on each row.

A natural question is how to decide to use something complicated like SPC(46—2*1). While some of it is just experience, most of it came from working through things by hand. For example, a table like the following one helps.

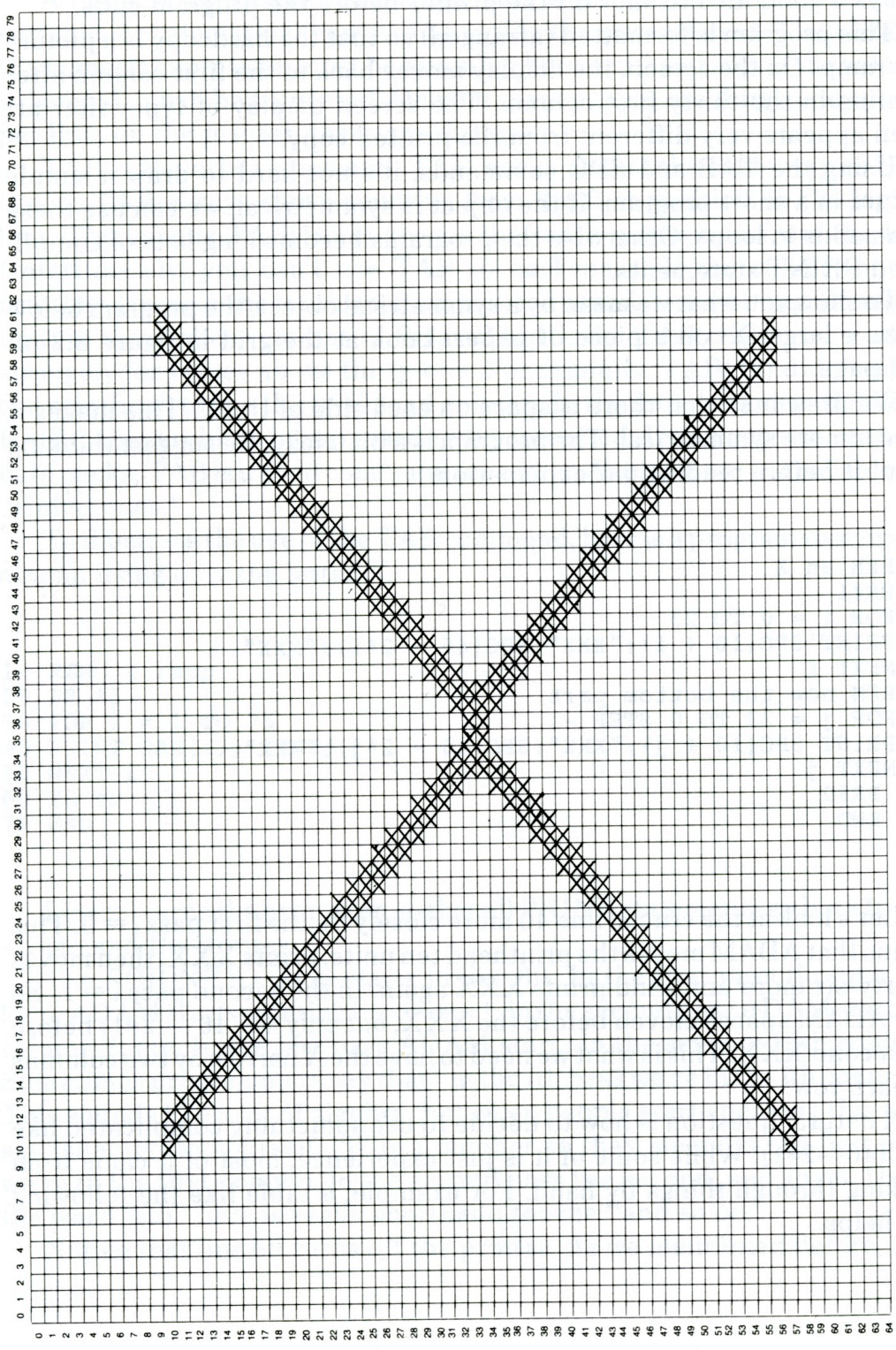


Fig. 8.2

Pass	Spaces on Left	Spaces on Right
1	10	46
2	11	44
3	12	42
4	13	40
	and so on	

Table 8.1

After this, it's not so hard to see that the first SPC should go up by one but the second should go down by two. Just remember that until you understand a pattern, you can't program ADAM to do it.

For our last printed drawing, we wanted to draw the word ADAM as it appears on the shipping box. In chapter 7, we had ADAM paint its name on the screen using low-res graphics—here we're getting much fancier. Before we could write this program, we traced the word ADAM on the shipping box (we had to use tracing paper to do it). Then we transferred it to graph paper as on the following page. To trace each letter so that it would look right we needed twelve rows of boxes. As an added embellishment we decided to use A's to print the letter *A,* D's for the *D,* etc. Then we counted boxes for quite a while. The result is the program on page 189. It is not a pretty program—it consists of twelve complicated PRINT statements—but it works. If you have the patience to type it in, the results are impressive. Expect to make several typing errors, just as we did. The programming was tedious but the results were worthwhile. This is especially so if you SAVE the program; then you can print lots of copies.

To translate this picture into a program, look across the top row in the figure on the following page. There are some spaces; then the top of the letter *A* takes up eight spaces on the grid, and then we have some more spaces and the top of the *D*. Count the spaces and count the blocks inside the letters, then enter a PRINT statement that matches the count. Line 40 in the program makes the printer type out the top row of the grid. Now do it for the second row—this corresponds to line 50. Continue in this fashion.

THE SPACES IN QUOTATION MARKS IN THIS PROGRAM ARE VERY IMPORTANT! BE CAREFUL ABOUT ENTERING THEM. THERE ARE EITHER ONE OR TWO SPACES.

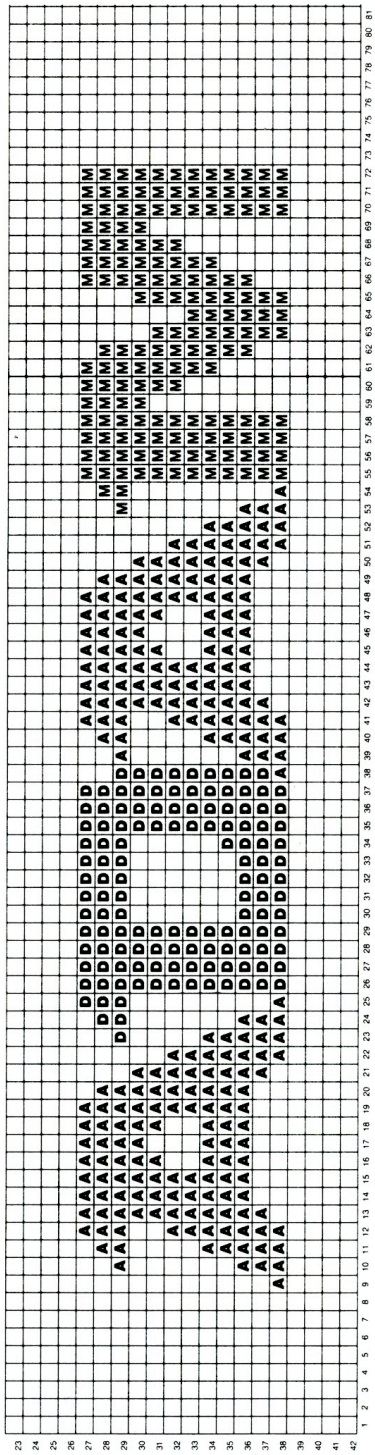


Fig. 8.3

Here's the program:

```
10 REM program to print the word ADAM
20 PR #1
30 FOR i = 1 TO 10: PRINT: NEXT i
40 PRINT SPC(7); "AAAAAAAA"; SPC(5); "DDDDDDDDDDDDD"; SPC(3);
"AAAAAAAA"; SPC(5); "MMMMMMM"; SPC(4); "MMMMMMM"
50 PRINT SPC(6); "AAAAAAAAAA"; SPC(3); "DDDDDDDDDDDDDDD";
SPC(1); "AAAAAAAAAA"; SPC(3);"MMMMMMMMM"; SPC(3); "MMMMMMM"
60 PRINT SPC(5); "AAAAAAAAAAA"; SPC(2); "DDDDDDDDDDDDDDDD";
SPC(1); "AAAAAAAAAA"; SPC(2); "MMMMMMMMM"; SPC(3);
"MMMMMMM"
70 PRINT SPC(8); "AAAAAAAAA"; SPC(4); "DDDD"; SPC(5); "DDDD";
SPC(3); "AAAAAAAAA"; SPC(3); "MMMMMMMM"; SPC(2); "MMMMMMMM"
80 PRINT SPC(8); "AAAA AAAA"; SPC(4); "DDDD"; SPC(5); "DDDD";
SPC(3); "AAAA AAAA"; SPC(3); "MMMM MMMM MMMM MMM"
90 PRINT SPC(7); "AAAA"; SPC(3); " AAAA"; SPC(3); "DDDD";
SPC(5); "DDDD  AAAA";SPC(3); "AAAA  MMMM MMMM MMMM MMM"
100 PRINT SPC(7); "AAAA"; SPC(3); "AAAA"; SPC(3); "DDDD";
SPC(5); "DDDD  AAAA"; SPC(3); "AAAA  MMMM  MMMMMMM  MMM"
110 PRINT SPC(6); "AAAAAAAAAAAAA  DDDD"; SPC(5);
"DDDD AAAAAAAAAAAAA MMMM  MMMMMMM  MMM"
120 PRINT SPC(6); "AAAAAAAAAAAAA  DDDD"; SPC(4);
"DDDDD AAAAAAAAAAAAA MMMM"; SPC(3); "MMMMM"; SPC(3);"MMM"
130 PRINT SPC(5); "AAAAAAAAAAAAAAA DDDDDDDDDDDDD";
"AAAAAAAAAAAAAAAMMMM"; SPC(3); "MMMMM"; SPC(3); "MMM"
140 PRINT SPC(5); "AAAA"; SPC(7); "AAAA DDDDDDDDDDDDAAAA";
SPC(7); "AAAAMMMM"; SPC(4); "MMM"; SPC(4); "MMM"
150 PRINT SPC(4); "AAAA"; SPC(9); "AAAADDDDDDDDDDDDAAAA";
SPC(9); "AAAMMMM"; SPC(4); "MMM"; SPC(4); "MMM"
160 FOR i = 1 TO 10: PRINT: NEXT i
170 PR #0
180 END
```

As you can see from this program, typewriter art is often a long collection of complicated statements. There are few shortcuts that can reduce the length. It takes a fair amount of time and patience to lay out a picture and then enter the print commands into ADAM. Once you've typed in this kind of program and are satisfied, be sure to SAVE it (and make a backup copy); then you'll never have to do it again. In doing typewriter art, you can use whatever symbols you want. Tracing paper may help in transferring the outlines of the image to graph paper.

Self-Check 1

A. Question

What is the difference between these two statements?

```
PRINT "HELLO";SPC(10);"GOODBYE"

PRINT "HELLO";TAB(10);"GOODBYE"
```

B. Programs

Write programs that draw pictures of the following:

1. A star.
2. A house.
3. A bird.
4. Anything that interests you.

Pretty Printing

If you are wondering about the section title, it happens to be the buzzword for doing fancy things on the screen.

ADAM's normal way of printing on the screen is light-colored symbols on a dark-colored background. Sometimes to jazz up a display (or a video game) you want to print in inverse or reverse video. *Inverse* or *reverse video* are the buzzwords for printing dark on light. The command that changes from the usual display to inverse video is INVERSE. After the command INVERSE, everything is printed in reverse, similar to a photographic negative.

Try the following program:

```
10 REM an inverse display
20 INVERSE
30 HOME
40 PRINT:PRINT:PRINT
50 PRINT "THE ADAM FAMILY COMPUTER"
60 END
```

The command that lets you print normally again is NORMAL. It can be used either in programming mode or in direct mode. It's easy enough to remember this but there is one caution: once something is displayed in reverse video, it stays that way. NORMAL just affects what follows.

ADAM has one other command to create fancy displays. The command FLASH makes ADAM's printing alternate between normal and inverse video. It could be used in a video game to give a flashing warning like:

```
100 FLASH
110 PRINT "TIME'S RUNNING OUT"
120 NORMAL
.
.
.
.
.
```

As before, the command NORMAL stops any further displays from flashing but has no effect on what has already been printed. We prefer displays using INVERSE to those using FLASH, but experimenting is fun. You could try:

```
10 REM which do you like better?
20 HOME
30 INVERSE
40 PRINT "THIS IS INVERSE"
50 NORMAL
60 PRINT: PRINT: PRINT
70 FLASH
80 PRINT "THIS IS FLASH MODE"
90 NORMAL
100 PRINT:PRINT:PRINT
110 PRINT "BACK TO NORMAL"
120 END
```

On some of the earliest SmartBASIC tapes, FLASH didn't work. If yours doesn't, we suggest writing COLECO or calling the toll-free number and getting a replacement.

In the last section we showed you ways to control the SmartWriter printer. Now we want to explain how to get better control of the screen. You can use the TAB and SPC commands from the last section. As we said, they work equally well on the screen. For example, suppose you want to display a table on the screen. If the heading of each column has to be five spaces apart, like:

```
HOURS      WAGES      NET
```

you would have to type:

```
PRINT "      ";"HOURS";"      ";"WAGES";"      "
"NET"
```

Even if you can always count spaces without making errors, it's a waste of time. Whenever there are more than two, we use the command SPC.

```
PRINT SPC(5);"HOURS";SPC(5);"WAGES";SPC(5);
"NET"
```

This prints the heading with the desired spacing starting from wherever the cursor was. To prevent problems use the HOME command right before this PRINT statement. Then the heading will appear at the top line of the screen.

TAB and SPC can be used to make intricate displays. A variable (for example, the counter variable from a FOR-NEXT loop) or a calculation can be used inside the parentheses. The value of this variable (or calculation) determines the spacing.

Look at the following program:

```
10 REM what does this program do?
20 HOME
30 FOR i= 1 TO 23
40 PRINT TAB(i);"$"
50 NEXT i
60 END
```

To see what this program does, check what happens when *i* equals one, two, and three. The first time ADAM goes through the loop, it indents one space and prints a dollar sign ($). Since there is no semicolon at the end of this PRINT statement, ADAM goes to the beginning of the next line (returns the carriage) before going on to the next statement. On the second pass, ADAM indents two spaces and prints a dollar sign, etc. The end result is a diagonal of dollar signs.

We haven't yet shown you a fast way to move down the screen. ADAM has two other commands that give you complete control of the cursor. You can think of these commands as "super arrow keys." The commands are VTAB and HTAB. For example, if you wanted to print HELLO on the twelfth line of the screen, you can type:

```
HOME:  VTAB(12):  PRINT  "HELLO"
```

For VTAB, the lines are numbered from 1 (the top) to 24 (the bottom).

HTAB is usually used with VTAB and often both appear on one line. For example, the commands:

```
VTAB(12):HTAB(15)
```

tell ADAM to move the cursor to the center of your screen. (Remember there are twenty-four lines and thirty-one columns when working with TEXT.) To design a screen display, you use graph paper with a twenty-four-column-high by thirty-one-column-wide grid.

The number inside the parentheses can be replaced by a numeric expression. The value of the variable or expression controls the cursor movement. If the number (or value of the numeric expression) for VTAB is less than 0 or greater than 24, you'll get an error message. If the value isn't a whole number then it is truncated. For example, VTAB(3), VTAB(3.2), and VTAB(3.9) all position the cursor on the third line.

The restrictions on the number (or value) for HTAB are different. It must be between 0 and 255. HTAB(1) is always the first position on the line where the cursor is. It is safer to use VTAB before HTAB because VTAB positions the cursor on whatever line you specify. HTAB(32) moves the

cursor down one row and positions it in the first column in that row. If you give the command HTAB(5.9) then, as before, ADAM truncates the 5.9, so that it is the same command as HTAB(5). Remember VTAB and HTAB are *never* used in PRINT statements and should be used *before* a PRINT command.

Here is a program that displays a frame built of stars and then prints something in reverse video in the center. The outline is simple:

1. Draw the top—thirty-one stars in a row.

2. Draw twenty-two lines each having a star in column 1 and 31.

3. Draw the bottom—thirty-one stars in a row.

4. Print whatever you want in the middle.

Unfortunately, this won't quite work. If you write a program to correspond directly to this outline you would get a box—but it would stay on the screen for less than one second! Then the top of the box would scroll (disappear) off the top of the screen. Remember that after a program ends the cursor always moves to another line and displays the]. The automatic carriage return together with the program as outlined would actually use up 25 lines, one more than ADAM's screen has. The result is that the top of the box is lost. Often you discover this kind of problem after you write a program. A slight change eliminates it. Here is the program:

```
10 REM a box of stars
20 HOME
30 FOR a = 1 TO 31
40 PRINT "*";
50 NEXT a
60 FOR b = 2 TO 20
70 VTAB (b): PRINT "*": HTAB(31): PRINT "*"
80 NEXT b
90 VTAB(22)
100 FOR c = 1 TO 31
110 PRINT "*";
120 NEXT c
130 VTAB(11): HTAB(4)
140 INVERSE
150 PRINT "THE ADAM FAMILY COMPUTER"
160 NORMAL
170 END
```

The first FOR-NEXT loop (lines 30–50) prints a row of stars along the top. The second FOR-NEXT loop (lines 60–80) prints two stars on opposite sides of each screen line. The final FOR-NEXT loop does the same thing as the first one but in the twenty-second row because of line 90. As soon as we show you subroutines (see chapter 12) this duplication can be avoided. Line 130 shows the power of VTAB and HTAB. These commands freeze whatever is on the screen and yet you can still reposition the cursor wherever you want. Lines 140–150 print the message in reverse video in the center of the screen. Line 160 returns the display to normal printing. Anytime you use INVERSE or FLASH, it is a good idea to turn it off when it is no longer needed.

(This may not come up as a frame on your screen because of overscan. It's easy to modify this program to fit any screen. Try it. You might have to display a slightly smaller frame.)

Because VTAB and HTAB give you absolute control of the cursor position, they can be used in many other ways. Here is a sampling:

1. You can replace something that is flashing or displayed in reverse video: Use VTAB and HTAB to position the cursor at the first flashing character and overwrite the text with something in normal mode. This means that you may have to print blanks by using PRINT " " if the new message isn't as long as the old one.

2. A timing loop used together with overwriting can make a message flash on and off. This will have a slightly different effect from the flashing video.

Self-Check 2

A. Questions

1. Where does the statement VTAB(12):HTAB(31) place the cursor?

2. Is there any difference between the commands:

```
VTAB(31):PRINT "$"
```

and:

```
PRINT TAB(31); "$"
```

3. When the cursor is in the first column, where do TAB(15) and SPC(15) send it? If the cursor starts off in the fifth column, where do they send it?

B. Programs

1. Write a program that will draw a smaller frame of stars.
2. Write a program to display a Y-shaped figure made of dollar signs.
3. Instead of a rectangular frame, write a program that displays some other geometric figure, like a triangle or pentagon.
4. Write a program that will display a warning in FLASH mode and then overwrite it in normal mode.
5. Write a program that displays a message for five seconds, erases it, displays it again, etc. First write one using the HOME command, then another which writes over the message with spaces. Why is the second program more useful than the first?

Backspacing, Overwriting and Quotes

In this section we want to show you some other ways to get control of the screen and the printer. We said that it is difficult to overwrite or print backwards. If you are like us, at this stage in learning about computers, this becomes a challenge rather than a warning. That's okay; in fact we think it's the best way to learn to use your ADAM. We hope by now that you are comfortable enough with ADAM so that you will try some things on your own. There are sure to be articles on BASIC that interest you in a computer magazine. If a program seems worthwhile and you haven't learned a command that is used in it, you can look at the reference guide. While there are a

few more programming techniques to learn before you've mastered BASIC, you're well on your way.

ADAM understands what you type because each key (or combination of keys) sends a numbered code to ADAM's central processing unit (you found out in chapter 2 that the CPU is ADAM's "brain") For example, the code number for " is 34. ADAM has a command, *CHR$*, which lets you bypass the keyboard. Up to now it has been impossible to have ADAM print quotation marks. Entering:

```
PRINT " " "
```

gets you an error message, not a printed quotation mark. However the statement:

```
PRINT CHR$(34)
```

tells ADAM to print the character whose code number is 34, so it prints a quotation mark. Another example is the number 16, which is the code for CONTROL/P. If you enter:

```
PRINT CHR$(16)
```

ADAM thinks you have hit CONTROL/P and prints the contents of the screen on the SmartWriter.

There are 256 codes used by ADAM. Some of them give strange symbols. If you want to see them, you can try the following program:

```
10 REM ADAM's character set
20 HOME: SPEED = 100
30 FOR i = 1 TO 255
40 PRINT i,CHR$(i)
50 NEXT i
60 SPEED = 255
70 END
```

This program is pretty straightforward. The only reason we set the speed at 100 is to slow the display down. Change it if it's not comfortable. The buzzword *character set* means the symbols ADAM has built in. On a machine used in Japan, the built-in symbols would be very different. Don't worry if you hear bells, whistles, or the printer starts doing strange things when you run the program—you're not harming ADAM.

We aren't joking about the bell. If you listen closely when you run the program, you hear a bell. While some of ADAM's sound capabilities are beyond the reach of a BASIC program (see the afterword), a bell is built in and can be turned on by:

```
PRINT CHR$(7)
```

Try this—it works equally well in direct mode.

Table 8.2 below summarizes some of the results of this program and gives some other information on the codes used by ADAM.

Code Number	Key (or Key Combination)	Code Number	Key (or Key Combination)
0	CONTROL/@	17	CONTROL/Q
1	CONTROL/A	18	CONTROL/R
2	CONTROL/B	19	CONTROL/S
3	CONTROL/C	20	CONTROL/T
4	CONTROL/D	21	CONTROL/U
5	CONTROL/E	22	CONTROL/V
6	CONTROL/F	23	CONTROL/W
7	CONTROL/G	24	CONTROL/X
8	CONTROL/H	25	CONTROL/Y
9	CONTROL/I	26	CONTROL/Z
10	CONTROL/J	27	CONTROL/[
11	CONTROL/K	28	CONTROL/
12	CONTROL/L	29	CONTROL/]
13	CONTROL/M	30	CONTROL/^
14	CONTROL/N	31	CONTROL/_
15	CONTROL/O	32	SPACE BAR
16	CONTROL/P	33	!

(continued)

Code Number	Key (or Key Combination)	Code Number	Key (or Key Combination)
34	"	71	G
35	#	72	H
36	$	73	I
37	%	74	J
38	&	75	K
39	'	76	L
40	(	77	M
41	)	78	N
42	*	79	O
43	+	80	P
44	,	81	Q
45	-	82	R
46	.	83	S
47	/	84	T
48	0	85	U
49	1	86	V
50	2	87	W
51	3	88	X
52	4	89	Y
53	5	90	Z
54	6	91	[
55	7	92	
56	8	93	]
57	9	94	^
58	:	95	_
59	;	96	`
60	<	97	a
61	=	98	b
62	>	99	c
63	?	100	d
64	@	101	e
65	A	102	f
66	B	103	g
67	C	104	h
68	D	105	i
69	E	106	j
70	F	107	k

(continued)

Code Number	Key (or Key Combination)	Code Number	Key (or Key Combination)
108	l	118	v
109	m	119	w
110	n	120	x
111	o	121	y
112	p	122	z
113	q	123	{
114	r	124	†
115	s	125	}
116	t	126	
117	u	127	DELETE

Table 8.2

The CHR$ command is powerful, but sometimes it's overdone. To say:

```
PRINT CHR$(65);CHR$(68);CHR$(65);CHR$(77)
```

rather than:

```
PRINT "ADAM"
```

is silly. However if you wanted to print "*ADAM*" then you would have to enter:

```
PRINT CHR$(34);"ADAM;CHR$(34)
```

The first 128 codes (from 0 to 127) given in Table 8.2 are fairly standard in microcomputers. The code is called the ASCII code (ASCII is the abbreviation for American Standard Code for Information Interchange).

The codes above 127 are used differently on different computers—and sometimes used differently on the same computer. ADAM uses these codes for the Smart Keys and also to print in reverse video. (For these codes and special symbols see Appendix A.)

In this chapter we want to concentrate on the codes below 33. These are the ASCII control codes. They are important because they carry the information that controls the screen display, the SmartWriter printer, and the data pack drives. In table 8.3, you can see what the control keys (or codes) do. We have listed only those that can be used in a program. Many of the codes will print unusual symbols on the screen with no effect on the printer. They may, for example, print card suits (diamonds, clubs, hearts), music and mathematical symbols, and even a sailboat. If you need any of these special symbols, you can find out their codes by running program 8.8.

CONTROL KEYS AND CODES*

Code	Key Combination	Effect on the Printer and Screen
1	CONTROL/A	
2	CONTROL/B	
3	CONTROL/C	
4	CONTROL/D	controls the data pack drives (see chapter 16)
5	CONTROL/E	
6	CONTROL/F	
7	CONTROL/G	rings the bell
8	CONTROL/H	backspace
9	CONTROL/I	tab
10	CONTROL/J	linefeed without carriage return
11	CONTROL/K	half linefeed without carriage return
12	CONTROL/L	clears screen (and repositions the cursor)
13	CONTROL/M	linefeed with carriage return
14	CONTROL/N	prints backwards (also see Interlude 1)
15	CONTROL/O	sends the printer back to forward printing (also see Interlude 1)
16	CONTROL/P	prints the screen
17	CONTROL/Q	

(continued)

18	CONTROL/R	
19	CONTROL/S	halts processing and printing; hitting any key restarts the process
20	CONTROL/T	
21	CONTROL/U	
22	CONTROL/V	
23	CONTROL/W	
24	CONTROL/X	
25	CONTROL/Y	
26	CONTROL/Z	
27	CONTROL/[& CONTROL/ ESCAPE	
28	CONTROL/	
29	CONTROL/]	
30	CONTROL/^	
31	CONTROL/_	
32	SPACE	

* For the symbols these codes display see Appendix A

Table 8.3

To obtain the effect given in the table, enter PRINT CHR$( ) and insert the code number of the effect you want inside the parentheses. For example, the table tells you that 14 is the ASCII code number that makes the printhead move backwards. Try the following program:

```
10 REM how to get boldface printing
20 PR#1
30 PRINT "MADAM";CHR$(14);" MADAM"
40 PR#0
50 END
```

This program not only overwrites and prints backwards, but it also prints the same characters in the forward pass and the backward pass. This is a good way to get the SmartWriter to highlight characters—it gives a boldface. Notice that we left an extra space inside the second pair of quotation marks. This is necessary when you overwrite—do you see why?

If you start the printhead running backwards, you'd better start it running the correct direction again. The command PRINT CHR$(15) tells the printer to print in the forward direction. We should add to the previous program (say, at line 35) the statement:

```
35 PRINT CHR$(15)
```

Here's another example. The ASCII for vertical tab is 10, so the line:

```
100 PR#1:  PRINT "HELLO";  CHR$(10); "GOODBYE"
```

makes the SmartWriter print:

```
HELLO
      GOODBYE
```

Self-Check 3

A. Questions

1. Using the ASCII code tables to decipher what the following program does.

```
10 REM ?
20 FOR i= 97 TO 122
30 PRINT CHR$(i);
40 NEXT i
50 END
```

2. Use the same trick as in program on page 202 to have the SmartWriter print your name in boldface.

B. Program

Write a program that uses a SPEED loop and the CHR$(7) to play bells at different speeds.

Summary

In this chapter, we showed you how to use the printer to draw "hard copy" pictures. We showed you how to gain complete control of the screen by using the control codes. You learned the following commands:

INVERSE	prints in reverse (dark on light).
FLASH	makes the screen display of text turn on and off.
NORMAL	turns off inverse and flash and returns the screen to the normal way of printing.
VTAB	moves the cursor up and down without changing anything on the screen.
HTAB	moves the cursor left and right without changing anything displayed on the screen.
CHR$	is used to print a character from ADAM's ASCII code list.
SPC	acts as a space bar starting from the current position of the cursor or printhead.
TAB	starts printing in the specified column.

BUZZwords

ASCII CODE

CHARACTER SET

INVERSE (REVERSE) VIDEO

INPUT

OUTPUT

PRINTHEAD



Games and Functions

In this chapter you will learn about the SmartBASIC functions:

RND INT

and other numeric functions built into ADAM like:

ABS SGN LOG EXP

SIN COS TAN ATN

You will also learn how to create your own functions using the command:

DEF FN

You'll see how the RND function, the pseudo-random number generator, can be used to create games and produce interesting pictures. We will use the INT function to round off and truncate numbers and show you how an automatic cash register decides how to give you change.

For people who need to make scientific computations, we explain how to use SmartBASIC to get any of the functions common in scientific calculators.

The RND Function – Games of Chance and Abstract Pictures

There are programs that let you play blackjack or backgammon or most any game of chance with a computer. In card games and most other games, the play is unpredictable—this is exactly what is meant by a game of chance. Computers are machines and the behavior of machines should be predictable. The reason games of chance can be played on computers is because all BASICs have a built-in command which makes the behavior of the computer seem random. *Random* is the buzzword meaning "by chance." Here's an example:

Enter (in direct mode):

```
PRINT RND(1)
```

ADAM will respond by displaying the number .732004777.

Now enter PRINT RND(1) again, and ADAM will respond:

```
.425420012
```

These numbers don't seem to have much to do with each other. Now try the following program:

```
10 REM  a demo of the RaNDom function
20 FOR i = 1 TO 10
30 PRINT RND(1)
40 NEXT i
50 END
```

After you enter RUN, ten numbers between 0 and 1 roll down the screen.

These numbers will seem to follow no pattern; that's what we mean by random. Each time ADAM processes a line containing the command RND(1) a different number between 0 and 1 pops out.

It's natural to wonder what a random number with nine decimal places is good for. Here's an example. Suppose we wanted to simulate (*simulate* is the buzzword for imitate) a coin toss. There are two possibilities—heads or tails. Since the numbers ADAM produces by repeated use of RND(1) don't follow a pattern, these numbers are just as likely to be less than .5 as bigger than .5. We tell ADAM that whenever RND(1) is less than .5 it should tell us that the coin came up heads. If RND(1) is at least .5, ADAM should say that the coin came up tails.

Here is a program that uses this trick to imitate a single coin toss.

```
10 REM ADAM tosses a coin
20 LET a = RND(1)
30 IF a < .5 THEN PRINT "It was HEADS"
40 IF a >= .5 THEN PRINT "It was TAILS"
50 PRINT "Do you want me to toss again"
60 INPUT "Enter yes or no "; s$
70 IF s$ = "yes" THEN 20
80 END
```

If you asked for 50,000 coin tosses and kept score, you would get very close to 25,000 heads and 25,000 tails.

As we mentioned before, the numbers you get by using RND(1) are called *pseudo-random numbers. Pseudo* means fake and, in theory, if you knew enough about ADAM then you could figure out the pattern. Don't let this worry you—it's also true for the mechanical card shufflers at Las Vegas. (When someone says something can be done "in theory," it doesn't mean that it's practical to do it. It might mean that it could be figured out in five billion years.)

In summary, the command RND(1) gives a number between 0 and 1 which is different each time it is used in a program. These numbers are

called pseudo-random numbers and can be used to simulate chance events.

Don't worry about the 1 inside the parentheses: it's just a place holder—any positive number would work as well. (The last section of this chapter explains more about this command and what the 1 does. That section isn't needed to use the RND(1) command.)

Numbers between 0 and 1 may be good for imitating a coin toss but not much else. There is a simple trick, called *scaling,* that lets you use this number in many other situations. Suppose you take a number between 0 and 1 that's always less than 1 (say .432657) and multiply it by 16. Now, it's between 0 and 16 but never quite 16 (6.922512 in our example).

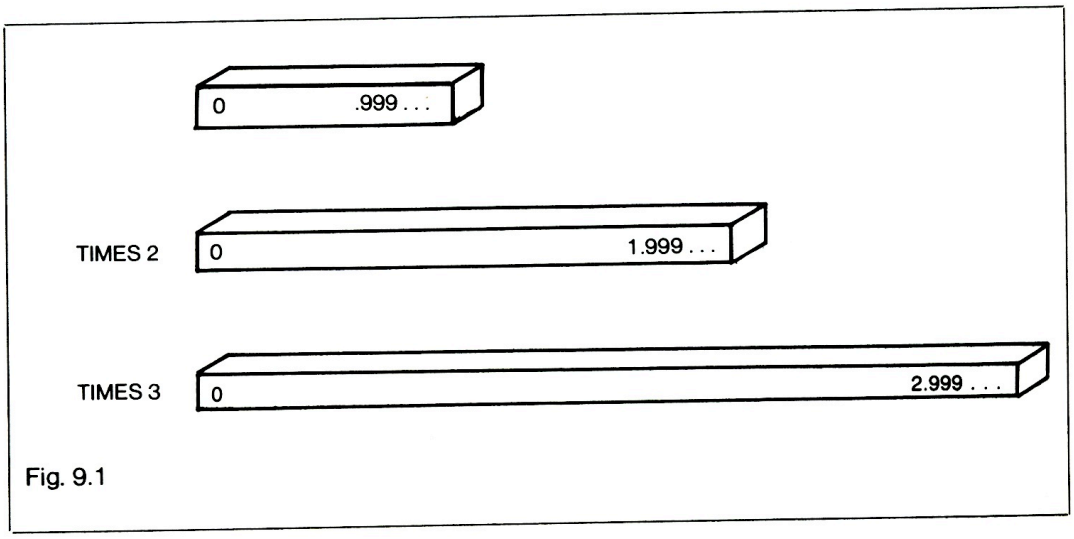


Fig. 9.1

If ADAM could truncate the number (chop off the decimal part) the result would be a number between 0 and 15. For example, if ADAM saw:

```
COLOR  = 6.489885  (or COLOR = 16*.432657)
```

ADAM would just assume we meant:

```
COLOR = 6
```

and continue executing the program.

Using this scaling trick ADAM can paint a row on the low-res graphics screen with randomly colored blocks. Here's the program to do it:

```
10 REM a randomly colored row
20 GR
30 FOR col = 0 TO 39
40 COLOR = 16*RND(1)
50 PLOT col,20
60 NEXT col
70 END
```

This program gives you a random design along the center of your screen. The most important line is line 40 which changes the color each time ADAM goes around the loop. The decimal part of 16*RND(1) is truncated (chopped off) and then the number that remains is an allowable color code—a number between 0 and 15. Line 50 then plots a block in that row in a new and unpredictable color.

Instead of painting row 20 alone, suppose ADAM went through every row and painted all the blocks using random colors. The entire graphics screen would then be filled with randomly colored blocks and form an attractive abstract design. Here's the program:

```
10 REM randomly colored blocks
20 HOME
30 PRINT "Do you want to see an abstract design?"
40 INPUT "Enter (a lower case) s to see it. ";s$
50 IF s$ < > "s" THEN 1000
60 GR
70 FOR row = 0 TO 39
80 FOR col = 0 TO 39
90 COLOR = 16*RND(1)
100 PLOT col,row
110 NEXT col
120 NEXT row
130 HOME
140 PRINT "Do you want to see a different design?"
150 INPUT "Enter yes or no ";yn$
160 IF yn$ = "yes" THEN TEXT: GOTO 60
1000 TEXT
1010 END
```

Let's go over this program piece by piece.

Lines 10–50 gives a user-friendly way to start the program.

Line 60 starts the graphics module. It uses a pair of nested FOR-NEXT loops to paint the screen. The outer loop starts at the top row and then the inner loop moves along each column. The inner loop is just like the program above. It moves from the first column to the last column. After ADAM finishes all the columns in one row, it begins painting the next row.

Line 90 is the crucial line. Since RND(1) is a number between 0 and 1, scaling (multiplying) it by 16 gives us a number between 0 and 16, but never equal to 16. After ADAM chops off the decimal part to obey the color command, the color will be randomly changed. This makes the picture interesting.

Lines 140–160 give you the option to see a different design. It will be different because the RND(1) command will give completely different numbers and therefore a different color pattern the next time it is used in line 90.

Lines 1000 and 1010 could easily have appeared on one line. It is good programming practice to have the END statement on its own line, so that it is easily found.

Self-Check 1

A. Question

What values does 100*RND(1) take on?

B. Programs

1. Write a program to generate (*generate* is the buzzword for create) fifteen random numbers between 0 and 25.
2. Write a program that generates fifteen random numbers between 1 and 32.
3. Using VTAB and HTAB and your programs from 1 and 2 above, write a program to display a random pattern of fifteen stars (*) on the screen.
4. Write a program to average fifty random numbers between 0 and 1.

5. One way to test a random number generator is to count how many of the numbers fall between 0. and .25, how many are between .25 and .5, between .5 and .75, and between .75 and 1. Write a program that generates 100,000 random numbers and counts how many fall into each of the four regions. (If you run this, don't worry if it takes a while to run.)

More on RND and INT

We told you that ADAM truncates numbers in certain commands: COLOR, TAB, SPC, VTAB, and HTAB; this lets you draw interesting designs. We still don't know how to imitate a pair of dice or a roulette wheel or even a number guessing game. A number with eight decimal digits isn't a very good roulette number. Because the truncation process is so important, ADAM has a built-in command to truncate all digits past the decimal point. The command is called INT. A number without anything after the decimal point is called an *integer.* INT is an abbreviation of the word integer. To see how INT works, enter:

```
PRINT INT(6.3456)
```

or:

```
PRINT INT(.98214)
```

INT is a *built-in* or *internal function* in almost every version of BASIC. You've actually seen lots of these functions so far. We've just avoided calling them by this name. A *function* is a command that takes something you

give ADAM and then does something to (or with) it. This isn't the formal definition of a function but it's a good way to think about it.

INT works differently with negative numbers.

$$\text{INT}(-3.5) = -4 \quad \text{not} \quad -3$$
$$\text{INT}(-6.1) = -7 \quad \text{not} \quad -6$$

INT is often called the "floor function" (it's technically called the *greatest integer function*) and if you think "floor," you have a good way to remember the rule for negative numbers. Here's how it works. Suppose you were building a house and had a machine that put in a foundation or floor automatically. However, this machine only does things in one foot divisions. You could dig one foot below ground or two or three but never one and a half feet. Similarly, you could build up one foot or two or three but never two and a half feet. This machine would work exactly the same way as the INT command. If you asked for a floor three and a half feet above ground you would get one three feet high. If you needed a foundation two and a half feet below ground (−2.5), the machine would dig down three feet (−3).

You can use the INT function, the RND(1) function, and scaling to get random whole numbers. PRINT INT(10*RND(1)) always gives a random integer between 0 and 9. PRINT INT(100*RND(1)) always gives an unpredictable integer between 0 and 99. PRINT INT(100*RND(1)) + 500 gives an unpredictable integer between 500 and 599.

Suppose you want to have ADAM play a number guessing game. Here's a program that can do it:

```
10 REM a guessing game
20 HOME
30 PRINT "Wait a second I'm thinking"
40 FOR p = 1 TO 1000
50 LET a = RND(1)
60 NEXT p
70 LET n = INT(100*RND(1))
80 PRINT "I'm ready - my number is between 0 and 99."
90 INPUT "Enter your guess. ";g
100 IF g > n THEN PRINT "TOO LARGE": GOTO 200
110 IF g < n THEN PRINT "TOO SMALL": GOTO 200
120 PRINT "That's right, you win.":
130 FOR i = 1 TO 10000: NEXT i: GOTO 1000
200 PRINT "Do you want to guess again?"
210 INPUT "Enter yes or no ";yn$
220 IF yn$ = "yes" THEN 90
1000 HOME
1010 END
```

Lines 30–60 are not included for dramatic effect. Every time you run a program, ADAM starts off with the same list of random numbers. A good analogy is to think that there is an incredibly long list in ADAM's memory filled with unpredictable numbers and ADAM always goes back to the beginning of the list for each new program. In the pause indicated on line 30, you have time to go through the first 1000 random numbers. The rest of the program then uses the numbers starting with the 1001st. In the last section we'll explain other ways to get around this problem.

The other lines of the program are similar to lines we have already used in other programs. Some of the graphic techniques from the last two chapters could jazz things up. By now you should start thinking about making programs more attractive, easier to use, and more useful. By using colors you can make programs more interesting.

Using INT and RND(1) gives numbers between 0 and the scaling number minus 1. That's why scaling by 100 gives a number between 0 and 99. If you want to simulate a pair of dice you need numbers between 1 and 6. To get a number with the right property you have to:

1. Scale by 6 to get an integer between 0 and 5.

2. Add 1 to that integer which gives an integer equal to 1, 2, 3, 4, 5, or 6.

Here is a dice simulation program:

```
10 REM a dice simulation
20 HOME
30 PRINT "Wait a second please"
40 FOR p = 1 TO 999
50 LET a = RND(1)
60 NEXT p
70 LET d1 = INT(6*RND(1)) + 1
80 LET d2 = INT(6*RND(1)) + 1
90 HOME: VTAB 12: HTAB 12
100 PRINT "FIRST DIE IS A ";d1
110 HTAB 12: PRINT "SECOND DIE IS A ";d2
120 HTAB 12: PRINT "THE TOTAL IS "; d1+d2
130 HTAB 12: PRINT "Shall I throw again?"
140 HTAB 12: INPUT "Enter yes or no. ";yn$
150 IF yn$ = "yes" THEN 70
160 HOME
170 END
```

This program is similar to the previous one. Because RND(1) changes each time, you might wonder when doubles occur. (Doubles are both dice

showing the same number.) This happens when the random numbers in lines 70 and 80 are close enough.

Suppose the random numbers are .732567812 and .698213412. They are scaled up to the numbers 4.395406872 and 4.189280472, and then, after truncating and adding one, you get double fives. Since these random numbers are supposed to behave erratically, they should be close enough for ADAM to get doubles about as often as to get doubles from unloaded dice.

The random number generator can also be used to create random letters or even random punctuation marks. To change numbers into characters we use the function CHR$ introduced in the last chapter. The random numbers have to be scaled so that they lie in the range we want. For example, lowercase letters have ASCII codes between 97 and 122; uppercase letters have codes between 65 and 90.

Everyone knows about "four-letter words." There are good ones like love and bad ones that we won't mention. They are actually very few four-letter words in the English language. Here is a program that will print random four-letter combinations until you stop it with CONTROL/C. See how many of them you recognize as English words:

```
10 REM random four letter combinations
20 PRINT "This program creates random four-letter
combinations."
30 INPUT "Do you want to see them? (Enter y or n.) ";yn$
40 IF yn$ <> "Y" AND yn$ <> "y" THEN 100
50 FOR j = 1 TO 4
60 PRINT CHR$(97 + 26*RND(1));
70 NEXT j
80 PRINT
90 GOTO 50
100 END
```

The little loop in lines 50 to 70 places four random lowercase letters next to each other on a line. Line 80 is included so that the next four-letter word is printed on the following line (and not right next to the previous word). Remember that this program is stuck in an infinite loop until it is stopped by CONTROL/C.

Self-Check 2

A. Questions

1. Compute INT(−47.2), INT(45), and INT(47.9).

2. Compute 11 − 11*INT(11/3).
3. Compute 17 − 2*INT(17/2).
4. What are the possible numbers you get displayed when you enter the following?

```
Print INT(1000*RND(1))
```

B. Programs

1. Write a program to simulate a roulette wheel. (There are thirty-seven numbers on a roulette wheel from 0 to 36.)
2. In most casinos there are two (or even three) zeroes. This increases the profit made by the casino. Write a program that simulates this kind of roulette wheel.
3. Write a program that displays "flash cards" for addition on the screen. (You want ADAM to generate two random numbers between, say, 2 and 12 and ask you to add them. After you enter the sum of the two numbers, ADAM should tell you whether your answer is correct.)

Other Uses for INT

INT is a useful function. Thus far we have only used it with RND. In this section, you will see several other uses for INT. For example, suppose you want to decide whether a number is odd or even. It's easy for you to do, but not so easy for ADAM. If you wanted to write a real roulette program, you would need a module to do this computation. When a number is even, dividing it by 2 leaves no remainder. If there is a remainder then the number is odd. We can use INT to test whether a number is odd or even by truncating the remainder and subtracting it from the original number. Here's the idea:

Take an odd number, like 5, and divide it by 2. You get 2.5. Using INT gives you 2. Because subtracting these two numbers doesn't give 0, we know 5 is odd.

```
10 REM an odd-even program
20 PRINT "Enter an integer"
30 INPUT "I'll tell you if it's odd or even. ";q
40 IF q/2 - INT(q/2) = 0 THEN PRINT q;" was even.": GOTO 10
50 PRINT q; " was odd."
100 END
```

Another use for INT is to round off a number instead of just chopping off the decimal part. Look at the following picture:



Fig. 9.2

The commuter wants to use the nearest station. There are several ways to find out which station is nearest, but our commuter is a bit eccentric. He'll find out which station is nearest by deciding which side of the halfway mark between the stations his house is located on. He just gets into his car and drives exactly one-half mile in the direction of the farther station. If he passes the station while driving, he knows that his house is closer to it than to the other station.

This may not be the easiest way to find the nearest railway station, but it is a good way for ADAM to find the integer closest to some number. The

round-off function is the imitation of this trick. The outline, together with the analogy with the commuter, is given in table 9.1.

THE COMMUTER	ADAM
1. starts with a house between 2 stations;	1. starts with a number between 2 integers;
2. drives exactly ½ mile in the direction of the farther station; and	2. adds .5 to the number; and
3. sees if he has driven past past the farther station.	3. sees if the sum has gone past the next integer (by using INT).

Table 9.1

You can see how the round-off function works by entering:

```
PRINT INT(6.2 + .5)
```

ADAM responds with a 6. However, to the command:

```
PRINT INT(6.6 + .5)
```

ADAM responds with the number 7.

In other books or in a programming class, you may see the following:

Round-Off Function = INT(x + .5)

The *x* probably reminds you of algebra, but if you just think of it as a placeholder then you won't have any problems. When using ADAM think of

x as being a number or the name of a variable. Another possibility you might see is:

Round-Off Function = INT(exprnm + .5)

This term *exprnm* is just another placeholder. It is an abbreviation in computerese that stands for a numeric expression which, as we said before, means either the name of a variable, the result of a calculation, or a number.

Though you don't need much (if any) algebra to program, it's a good idea to get used to this placeholder notation. It both saves space and helps in writing statements clearly and concisely.

Here's an example of using the round-off function. Suppose you wanted to program a cash register to give change. (We are not joking—some modern cash registers "speak" BASIC). Usually you want to give out the least number of coins and bills. Before we show you the program we want to work through an example by hand. (Imitating a computer by working through examples is one way to decide how to write a program.)

Suppose you have to give ninety-three cents in change. If you want to use the fewest coins, you should give:

3 quarters	=	.75
1 dime	=	.10
1 nickel	=	.05
3 pennies	=	.03

Before you continue reading, we suggest that you stop to think about how you know that you should give out three quarters. Remember, until you can analyze how you decide to do something, you can't program ADAM to do it.

You probably said "three quarters is seventy-five cents, which is too small, but four would be too much," and you don't ever want to give too much change. So you'd give three quarters as part of the change. You still have eighteen cents of change to return, so you continue with dimes. You might try programming ADAM to imitate this kind of reasoning. It's possible but hard. There is a trick that simplifies the programming.

Dividing 93 by 25 gives 3.72. Since you can't give .72 quarters, you want to give 3. The number 3 is just INT(93/25).

Now we can write the program:

```
10 REM  a program to make change
20 HOME
30 PRINT "Please enter an amount less then $1."
40 INPUT "no decimals please. ";c
50 LET q = INT (c/25)
60 LET c = c - 25*q
70 LET d = INT(c/10)
80 LET c = c - 10*d
90 LET n = INT(c/5)
100 LET c = c - 5*n
110 LET p = c
120 PRINT "The number of quarters is "; q
130 PRINT "The number of dimes is "; d
140 PRINT "The number of nickles is "; n
150 PRINT "The number of pennies is "; p
160 END
```

In lines 60, 80, and 100, ADAM changes the amount *c* by subtracting the amount of change already given. If we gave out three quarters, then line 60 makes the new *c* seventy-five cents less than the old.

(If you don't like all this typing, here's a trick you can use. Enter the following line:

```
115 LET a$ = "The number of  "
```

and then use it in lines 120–150. Line 120 becomes:

```
120 PRINT a$; "quarters is"; q
```

and the others have a similar form.)

Self-Check 3

A. Question

How do you find the remainder after dividing the following numbers by 5?

1. 21
2. −21
3. 149, 854

B. Programs

1. Write a program that finds the remainder after division by 5.
2. Write a program that finds the remainder after dividing any number by another number. What happens if the second number is 0?
3. Change the "change program" so that it works for amounts over twenty dollars.

ADAM's Built-In Functions

If you don't need scientific functions for school or fun you might want to skip this section. In programming a computer like ADAM, there is nothing wrong with skipping anything remotely "algebraic" and concentrating on its other uses. The format for this section is different; there are no programs. Mostly, we'll just explain each function. To do this effectively we need to use the placeholder notation. Our placeholder will be *exprnm.* As we mentioned earlier, this is computerese for a numeric expression—something ADAM can understand as a number. It can be a calculation or a variable name or just a number. ADAM always does the calculation inside the parentheses before doing the computation called for by the function.

ABS(exprnm) gives the "absolute value" of whatever is inside the parentheses. All this function does is remove minus signs. For example:

```
PRINT ABS(-3)   gives 3
PRINT ABS(3)    also gives 3
```

SGN(exprnm) gives the sign of the expression inside the parentheses. For example:

```
PRINT SGN(112)    gives 1
PRINT SGN(0)      gives 0
PRINT SGN(-37)    gives -1
```

SQR(exprnm) gives the positive square root of the numeric expression inside the parentheses. As examples, try:

```
PRINT SQR(100)    gives 10
PRINT SQR(2)      gives 1.41421356
```

The value ADAM gives for the square root of two is correct to nine decimal places but is not the exact answer. (The exact answer requires infinitely many decimal places.)

Caution: If you ask ADAM to compute the square root of a negative number, you'll get an error message and the program will bomb.

LOG(exprnm) gives the natural logarithm of exprmn. If you haven't seen the natural log then forget this function. Using the natural logarithm, it is very easy to get "common logs," logarithms to the base ten. The formula is given in Appendix C. This formula can be made into a user-defined function (see the next section).

EXP(exprnm) raises the base *e* (approximately 2.718281828) of the natural logarithm to the power of whatever is in the parentheses. (If you felt like typing in the nine-digit number *e* you could use the exponentiation symbol ^).

ADAM's trigonometric functions

Like many calculators, ADAM uses what is called *radian measure* instead of degree measure for the size of angles. Radians are used in science and math because they simplify many formulas. In radian measure, one full revolution is 2π radians. In degree measure, it is 360 degrees. To translate other measurements in degrees into radians, you can use proportions. The formula is:

$$\frac{\text{size in degrees}}{360} = \frac{\text{size in radians}}{2\pi}$$

One radian is about 57.3 degrees.

There are three trigonometric functions and one inverse trigonometric function built into ADAM. All trigonometric functions can be formed using only these three functions. They are:

SIN(exprnm) gives the sine of the angle *exprnm. Exprnm* is calculated to give a number which ADAM uses as the number of radians. *Exprnm* can be any legal numeric expression. For example:

```
a = SIN(0)
```

places the number 0 = sin(0) in the box labelled *a.*

COS(exprnm) gives the cosine of the angle *exprnm* measured in radians. *Exprnm* can be any legal numeric expression. For example:

```
a = COS(0)
```

sets the value of the contents of the box *a* equal to cos(0) = 1.

TAN(exprnm) gives the tangent of the angle *exprnm* measured in radians. *Exprnm* can be any legal numeric expression. For example:

```
a = TAN(pi/4)
```

would place the number 1 in the *a* box if *pi* is the number π. There is an easy way to enter the value of π (see Tricks of the Trade following this section).

ATN(exprnm) gives the arctangent of the numeric expression *exprnm.* The arctangent gives answers between $-\pi/2$ and $\pi 12$. For example:

```
a = ATN(1)
```

places the value $\pi/4$ in the box *a.*

In Appendix C, there is a list of formulas for trigonometric and other mathematical functions which can be derived from the built-in functions.

If you need the number π then you can begin a module with the statement:

```
pi = 4*ATN(1)
```

with some line number. This gives you π to nine decimal places. (Remember not to use any other variable whose first two letters are *pi*.)

You can use a similar trick to change degree measure to radian measure.

```
rad = (2*4*ATN(1)/360)*deg
```

where *deg* is the size of an angle in degrees and *rad* is the size of the same angle in radians.

Tricks of the Trade: ADAM is only accurate to nine digits. This is *usually* more than enough accuracy. Sometimes it causes problems. We'll explain some of them.

When we write numbers, we use powers of ten and decimals—ADAM doesn't. Instead it uses powers of two. As a result, ordinary decimals like .1 and .7 can't be stored in ADAM's memory. ADAM stores numbers which are very close, but not exactly the same as them. Our next program illustrates this point dramatically.

```
10 REM a demo of round-off error
20 i = -1
30 If i = 0 THEN 70
40 i = i + .1
50 PRINT i
60 GOTO 30
70 END
```

This program is simply an infinite loop with an exit test and is like many of the programs in chapter 5. It starts with -1 and adds .1 to it on each pass through the loop. It ends if $i = 0$. You might think that ADAM will make ten

passes through the loop and then the program will end. Run it—you'll see that it is really an infinite loop. The test on line 30 is never successful. The problem is that ADAM can't represent .1 accurately. When it keeps adding its approximation to .1, it misses 0 by a very small amount. The test on line 30 is successful only when *i* is *exactly* zero and this never happens.

One other flaw of ADAM's decision to use powers of two for arithmetic shows up in its handling of, for example, INT(3.00). If you enter PRINT INT(3.00), ADAM will display the number 2 and not the 3 that you expect. This happens because the binary approximation to 3.00 that ADAM uses is slightly less than 3. This flaw is minor and doesn't effect INT(3) or INT (3.0000) for example. In any event it should be corrected in future versions of SmartBASIC.

Another type of problem is that in the course of running a program, ADAM may repeat arithmetic operations, like multiplication and addition, thousands of times. Even if ADAM is only off by the tiniest of fractions in the calculations, the errors can build up. This type of error is called *round-off error.* Manufacturers of large computers try to avoid this problem by making their machines accurate to twenty, forty, and sometimes even hundreds of decimal places. Even with this incredible accuracy, programmers still must be careful of round-off errors.

We didn't include this discussion to scare you (almost nothing you ask ADAM to do will be affected by round-off error) but, to master ADAM, you should be aware of the possibility.

User Defined Functions

In the last section we showed you some of the built-in functions that ADAM supports. (*Support* is the buzzword for has available.) Perhaps you weren't impressed. After all, some inexpensive ($25–$50) scientific calculators have as many (or more) built-in functions. While this is true, with SmartBASIC you can define your own functions. You can create as many functions as you like inside a program.

Suppose you need to round off many numbers in the course of a program. For example, to make bar graphs less optimistic and more accurate, put the line:

```
DEF FN rd(x) = INT(x + .5)
```

early in your program. Then, instead of using INT(37.3 + .5), you could use:

```
FN rd(37.3)
```

The DEF is supposed to remind you of the word *define* and the FN to remind you of the word *function.* The FN must come before the name of the function and the DEF FN statement must come before the function is used. The name of the function can be any legal variable name. (Remember that only the first two characters matter.)

User-defined functions are most important when one formula must be used many times in a program. Suppose you were doing physics homework that involved Einstein's formula for mass to energy conversion.

$e = mc^2$ (in computerese e = m*c*c or e = m*c ^ 2

Unfortunately, *c* is the speed of light in centimeters per second. This happens to be:

2.99793E10

and it isn't much fun to have to write:

e = m * 2.99793E10 ^ 2

every time you need it.

Instead, we enter the following:

```
DEF FN e(m) = m*2.99793E10^2
```

As before, the *m* is just a place holder. You can replace it with any number, variable, or calculation. ADAM always uses the value of what's inside

the parentheses. Then saying *FN e* (454) would give you the energy contained in 454 grams (1 pound).

Here's a programming example.

In algebra you often have to make up a table giving values of functions for different numbers. When we first learned about straight lines, we often had to make up these tables. (In chapter 15, we'll have ADAM draw all sorts of lines and curves, but teachers often want tables.) Suppose we have the line:

$$y = 3x + 7 \text{ (in computerese } y = 3*x + 7)$$

and we want to have ADAM print out a table like:

x	y
1	10
2	13
−1	4

and so on.

A program to do this is easy.

```
10 REM a table
20 HOME
30 DEF FN y(x) = 3*x + 7
40 HTAB (10): PRINT "  x      |    y "
50 FOR a = -10 TO 10
60 HTAB(10): PRINT "-----------------"
70 HTAB (12): PRINT a;"    |     ";FN y(a)
80 NEXT a
90 END
```

In fact, this would work for any straight line by adding a line 25:

```
25 INPUT "enter m and b separated by commas"
   ;m,b
```

and changing line 30 to:

```
30 DEF FN y(x) = m*x + b
```

(If you do this, be careful not to use *m* and *b* as variables anywhere else in the program.)

Self-Check 4

A. Questions

Define a function that:

1. Gives 1 if a number is odd and 0 if the number is even.
2. Gives the remainder after dividing by 10.
3. Converts radian measure of angles to degree measures.

B. Programs

Write programs to prepare tables for the following functions:

1. $\sqrt{1 - x*x}$
2. y = ABS(SIN(x))

More on RND

This section isn't necessary. In fact unless you feel comfortable with everything we've said so far we suggest skipping it! If you want to do any serious programming of games of chance it will prove useful.

We said RND(1) gives a pseudo-random number. How they are generated (there isn't a list in ADAM's memory) requires a fairly lengthy discussion and much more mathematics than we can give here. A *pseudo-random number* is one that has many of the properties of numbers obtained by chance. We used one property in the coin toss simulation—that the numbers should just as likely be less than .5 as greater than .5. The same thing

works for any other range. If you had ADAM calculate fifty thousand RND(1)s, then about 5000 would be between 0 and .1, 5000 between .1 and .2, and so on. Using RND(1) properly won't "fix" your games.

The key word, of course, is *properly*. We indicated one problem: ADAM always starts with the same number (.732004777) the first time RND is used in a program.

One way to get around this problem is to start your game programs with a module like:

```
10 REM Cutting the cards
20 PRINT "Everyone who wants to play"
30 PRINT "should press a number key."
40 INPUT "Then hit return. "; p
50 PRINT "We'll shuffle the cards for about "; p/250;
"seconds"
60 FOR i = 1 TO p
70 LET a = RND(1)
80 NEXT i
          .
          .
          .
```

This (so to speak) advances you in the list to a place that no one could know beforehand. Line 50 isn't necessary, but it saves worry about whether there is a mistake in the program. We know that a timing loop takes about one second for each 1000 passes. We can have ADAM calculate about how long the wait will be. (We got the number 250 by timing this loop; 1000 passes through the loop takes about four seconds.) There is another way to "cut the cards" that involves no delay; we'll show it to you soon.

The RND function can be affected by what you put inside the parentheses. If you say RND(0), ADAM always gives back the last random number generated. For example, if you enter (in direct mode):

```
PRINT RND(1)
```

ADAM will respond by displaying .732004777. If you then immediately give ADAM the command:

```
PRINT RND(0)
```

it will again give you the number .732004777. This is useful in trying to debug a program. (Imagine trying to check the way a program works if an important number changes each time you run it.) RND(0) gives a way of checking what random number ADAM has just used in a calculation.

More important is what happens when there is a negative number inside the parentheses. Each time you give the command RND (negative number) ADAM *reseeds* the random number generator. The buzzword for the (negative) number inside the parentheses is a *seed.* (The seed from which random numbers grow.) A good way to think of the seeds is that there is a different list of pseudo-random numbers for each (negative) seed. (For example, the lists corresponding to RND(−23) and RND(−1) are quite different.) This gives another way to "cut the cards." It also avoids the delay involved in advancing forward in the list, as was done in program 9.12. Another advantage to this reseeding process is that it changes the patterns given by RND(1). After reseeding, you can use RND(1) without worrying about the pattern being known in advance. This allows a "cut the cards" module of only four lines:

```
10 REM another way to cut the cards
20 PRINT "Everyone should press a number key"
30 INPUT "Now press enter ";p
40 LET a = RND(-p)
```

(This module takes almost no time to run—the earlier module may run for several minutes.)

Summary

In this chapter, you learned about the functions that are built into ADAM and how a programmer can build his or her own. ADAM's many built-in scientific functions and its ability to create new functions make it more versatile than a scientific calculator.

The pseudo-random number generator, built into ADAM, can be used to simulate random events, like throwing dice and roulette wheels. The function INT is useful in rounding off numbers.

BUZZwords

BUILT-IN FUNCTION

FUNCTION

GREATEST INTEGER FUNCTION

INTEGER

INTERNAL FUNCTION

PSEUDO-RANDOM

RADIAN MEASURE

RANDOM

RESEED

ROUND-OFF ERROR

ROUND-OFF FUNCTION

SCALING

SEED

SIMULATE

SUPPORT

USER-DEFINED FUNCTION



Lists and Sorting

In this chapter we'll show you how to use lists. You'll learn to use the commands:

DIM	DATA	READ	RESTORE
STOP	CLEAR	RUN line number	

and a new way to name and assign values to variables.

The two methods of assigning values to variables, discussed in earlier chapters, are:

1. a LET (assignment) statement; and

2. an INPUT statement.

With both of these methods, the variables were named and their values assigned at (essentially) the same time. The method shown in this chapter is different: We set aside space in ADAM's memory that can be filled in later. Variables can be named and used in a systematic way.

Up to now we've stressed that well-written programs should be friendly to the user. It's time to start getting friendly to the computer, too. You might wonder why it's necessary to be nice to ADAM—after all, it doesn't have feelings and it won't make intentional errors out of spite.

All this is true, but computers have natural ways of functioning. The closer our programs are to the way ADAM works, the faster they will run. For example, computers like things to be as organized as possible, so our programs should be as organized as we can make them. In any case, programming errors are less likely if you are well-organized. Computers excel at repeating things, so this is also a *computer friendly* operation. Repetitive operations are especially important since these can usually be written in just a few lines; for instance, using a FOR-NEXT loops.

Lists

Suppose we are writing a program which requires one variable with the value 1, a second with the value 2, a third with the value 3, and so on for 50 different variables. This kind of outline cries out for a FOR-NEXT loop something like:

```
FOR i = 1 to 50
Assign i to its variable
NEXT i
```

Of course, this won't work because we don't (yet) have a way of placing new variables in the body of the loop where the assignments are taking place.

Here's another example. Suppose we had to average 100 numbers. The program is easy:

```
10 REM the first averaging program
20 s = 0
30 FOR j = 1 TO 100
40 INPUT x
50 LET s = s+ x
60 NEXT j
70 PRINT s/100
80 END
```

However, after each pass through the loop ADAM loses the value of *x*. If the values were needed in some other part of the program, it wouldn't be much fun to have to enter them again.

Our final example is the bar graph program (refer to page 168). We had variables named *m1, m2, m3.* Suppose you wanted to write the input module for a twelve-month period using a FOR-NEXT loop like:

```
FOR i = 1 to 12
INPUT mi
NEXT
```

If this were possible it would be easy to write a bar graph program for any twelve-month period. Unfortunately, this is impossible. In all versions of BASIC *mi* is a perfectly good variable name—for a single variable. No BASIC lets you separate the *m* from the *i.*

After we show you a systematic way to name variables, it will be easy to solve these problems.

Everyone uses lists. You might make lists of your homework assignments, lists of things to buy when you go shopping, and lists of things not to forget. Sometimes lists are numbered. For example, suppose before going out you make a list of where you must go. The list might look like this:

Places

1. dry cleaners
2. supermarket
3. drugstore
4. shoemaker
5. department store
6. post office

If a copy of this list was left at home, you could call and say: "I'm at the drugstore" or "I'm at the third place" or "place number 3." The post office is place number 6, or the last place, etc. Suppose you were incredibly well-organized and knew you could visit at most six places each day. Then you might print some blank lists with each one having six slots ready to be filled in. Each day you could take out a new sheet and fill in the slots. These blank lines are exactly like the boxes (variables) that can be filled with values in ADAM's memory. *Drugstore* now occupies the third slot on today's "places" list.

To ADAM a list is just a collection of variables, each one of which is identified by two things:

the name of the list; and

its position on that list.

In SmartBASIC, lists are written in a slightly different form. ADAM uses *places$*(3) for the third item on the list of places. Notice the dollar sign (**$**) that indicates the variables will hold words (strings). The number in parentheses is called a *subscript.* The subscript (what's inside the parentheses) is important. To ADAM, *m5* is the name of a single variable but *m*(5) is the name of the fifth variable on a list. The term *subscript* comes from mathematics, where the number is often written slightly below the word. Here's an example of the mathematical way of writing subscripts:

$$\text{places}_3$$

If you ever have to translate mathematical formulas involving subscripts into SmartBASIC just follow this pattern.

There is only one more thing to mention before you can use lists. Lists can't be open-ended in any version of BASIC. SmartBASIC doesn't limit the possible length of your lists. It does require that you tell ADAM how long your list could be. Fortunately, it is never necessary to say exactly how long the list is. Just make sure that it is at least as long as the number of items you want it to contain. The statement that gives the length of a list is called the *dimension* statement and the command is DIM. For example, if someone never went to more than 20 stores, he or she would enter:

```
30 DIM places$(20)
```

Notice that we used line number 30 rather than some larger line number. This is because a dimension statement, which gives the size of a list, must come before any statement that mentions the list. The easiest way to remember to do this is to have dimension statements come near the beginning of the program (or near the beginning of the module in which the list is used). Values can be assigned to the variables *places$(1), places$(2),* etc., using the following fragment:

```
        .
        .
        .
200 DIM places$(20)
210 PRINT "In order, the places I will shop at today are"
220 PRINT "(Hit the return key to stop.)"
230 FOR i = 1 TO 20
240 PRINT "Number ";i;" is "
250 INPUT places$(i)
260 IF places$(i) = "" THEN 280
270 NEXT i
        .
        .
        .
```

This program has one new programming trick. Line 220 tells the person running the program to hit RETURN to stop. No other key has to be hit before the RETURN key. When the RETURN key is hit without anything typed before, nothing is entered into ADAM's memory. Line 260 asks whether *places$(i)* = "". Since there is nothing between the quotation marks, ADAM checks to see that nothing has been entered. This is a way of testing whether the RETURN key alone has been hit.

Now we have a computer-friendly way to input data into ADAM. It is also user-friendly. The PRINT statements tell us exactly what to type in when we receive the prompt **?** from the INPUT command.

For a new input module in the bar graph program you could use:

```
20 DIM m(12)
30 FOR i = 1 TO 12
40 PRINT "Sales in month # ";i;"  ";
50 INPUT m(i)
60 NEXT i
```

Because this operation fits ADAM's natural way of functioning, ADAM makes it easy for you to use short lists. If a list has no more than ten items, you don't need to use a dimension statement. As soon as the computer sees that there is a subscript on a variable, it automatically assigns ten slots, numbered from 1 to 10, to that list. The buzzword for anything that is automatically assigned to a command (or variable), unless you say otherwise, is the *default value* of that command (or variable). We say that the default value of the dimension of a list is ten. In a list made up of strings (words), each item can use up to 255 memory locations. So, using the default dimensioning, a string list *could* use 2550 memory locations. (Luckily, many fewer locations are usually needed.) Even though ADAM is an 80K machine (see chapter 2 if you don't remember what this means), only about 25,000 locations are available for your SmartBASIC program and variable storage. Each undimensioned alphanumeric list could use 10% of that space! This is a good reason for using dimension statements even for short lists. (Don't worry too much about the space taken up by lists of strings. SmartBASIC uses some tricks to save space.)

Dimension statements also give an easy way to find out the names of your lists. This is handy when modifying a program. In general it's a good idea to dimension all subscripted variables.

The programs mentioned in the beginning of this section shouldn't cause any problems now. Here's another one for finding an average:

```
10 REM averages and more
20 DIM  x(100)
30 LET s = 0: REM initialize the sum
40 FOR i = 1 TO 100
50 INPUT "Enter the next number "; x(i)
60 LET s = s + x(i)
70 NEXT i
80 PRINT "The average is "; s/100
90 PRINT "Do you want to see the numbers you entered?"
100 INPUT "Enter y or n ";yn$
110 IF yn$ = "n" THEN 160
120 HOME
130 FOR i = 1 TO 100
140 PRINT x(i)
150 NEXT i
160 END
```

Lines 90 to 150 were added to show you that ADAM's memory still holds the numbers you entered. This kind of module is often important. You may want to check that the numbers were entered correctly.

The program that assigns the first fifty numbers to fifty variables is no harder to code. (*Coding* is a buzzword we haven't used for a while. Recall that it means writing a program.) It can be jazzed up by letting the dimension statement be set from the outside. Using a variable or a calculation to dimension a list is called *dynamic dimensioning.*

```
10 REM an example of DYNAMIC DIMENSIONING
20 HOME
30 INPUT "How many variables? "; n
40 DIM z(n)
50 FOR q = 1 TO n
60 LET z(i) = q
70 NEXT q
   .
   .
   .
```

We've used *q* for the counting variable and *z(q)* for the list as a change of pace. There is no reason to stick to *i* and *x(i)* in all your programs.

Suppose we want to prepare a questionnaire to be given to twenty people. In order to use a list, each person is assigned an identification or i.d. number. The list with the responses to question one will be called *a1*. The The responses to question two will be *a2*, etc. If the response is given in words, we'll attach a dollar sign to the name of the list. A list of the numbers assigned to the people responding to the questionnaire could be held in *a1*. The names of the people responding could be kept in *a2$*. The number assigned to the fifth person answering the questionnaire would be *a1(5)* and *a2$(5)* is his or her name. Here is a program that asks people to fill out a questionnaire:

```
10 REM this program is a sample questionnaire
20 REM a1 is the list of assigned numbers
30 REM a2$ is the list of names
40 DIM a1(20),a2$(20),a3(20),a4(20)
50 FOR i = 1 TO 20
60 HOME
70 INPUT "What number were you assigned? ";a1(i):HOME
80 INPUT "What is your name? "; a2$(i)
90 PRINT "      Please type in the number"
100 PRINT "      on the list"
110 PRINT "      for your age group"
120 PRINT "      1.  0 to 10"
130 PRINT "      2. 11 to 20"
140 PRINT "      3. 21 to 30"
```

(continued)

```
150 PRINT "      4. 31 to 45"
160 PRINT "      5. 46 to 60"
170 PRINT "      6.  over 60"
180 INPUT a3(i):HOME
190 PRINT "    Please type in the number"
200 PRINT "    on the list for your usual"
210 PRINT "    transportation"
220 PRINT "          1. train"
230 PRINT "          2. bicycle"
240 PRINT "          3. automobile"
250 PRINT "          4. bus"
260 PRINT "          5. other"
270 INPUT a4(i):HOME
280 HOME: PRINT "Next person please."
290 PRINT "Enter q to quit or return  to continue"
300 INPUT q$
310 IF q$ = "q" THEN 330
320 HOME: NEXT i
330 HOME
```

This program is still not complete. There's no END statement because collecting information is silly if it isn't used. The reason for using lists is to keep, to rearrange, and to use the data collected. Without lists, the data entered by one person would write over the responses given by the previous person. Most of the data collected would be lost.

For example, suppose someone wanted to know how many people use bikes for most of their transportation. A person who uses a bike would have answered the third question with a 2. The following module counts the number of these responses:

```
500 j = 0
510 FOR i = 1 TO 20
520 IF a3(i) <> 2 THEN 540
530 j = j + 1
540 NEXT i
550 PRINT "  the number of people"
560 PRINT "  riding bikes is "; j
```

Someone wanting different information from this questionnaire would just write a (slightly) different module.

There are three important programming techniques in our program. The first is that one dimension statement can dimension several variables. The next is that each PRINT statement tells ADAM to print a very short

string. This allows the information to be printed on one line on the screen. It is annoying when words are split in the middle because they don't fit on a screen line. While this increases the number of lines in the program, it makes using the program easier. With questionnaires this is especially important because people won't fill out an unpleasant questionnaire.

Line 500 is important, although all it does is assign the value 0 to the variable named *j*. However, without this line the first mention of the variable *j* in this module would be in line 530 and that line doesn't say what value *j* had; the new value is just one more than the previous value. In a long and complicated program, *j* might already have been used somewhere else. In that case, its value isn't likely to be zero and the module wouldn't work correctly. Line 500 says, in this module, we want *j* to start off with the value zero. We strongly suggest that, if you don't want your long programs to run wild, be careful of this problem. Otherwise, be aware that it is one of the hardest bugs to locate and eliminate from a program.

> THE FIRST (EXECUTABLE) STATEMENT IN A MODULE WHICH USES A VARIABLE SHOULD ASSIGN THAT VARIABLE A DEFINITE VALUE.

As we said in chapter 6, the buzzword for assigning a definite value to a variable is *initializing* that variable. Line 500 initializes the variable *j*. Initializing makes sure that a variable has the right initial (beginning) value each time it is used for a different purpose in a program.

SmartBASIC does some initializing on its own. When ADAM begins running a program, SmartBASIC sets aside a box for each variable and fills that box with the value zero for boxes that will hold numbers, and the empty word for boxes that will hold strings. (The *empty word* is the string with no characters.) We used it in the program at the bottom of page 235. It's exactly what ADAM puts in its memory if you hit return without typing anything else. The empty word is like zero. A box could be filled with the empty word by the assignment statement:

```
LET a$ = "  "
```

Since there is nothing between the quotes, the box is filled with nothing. Sometimes people use the buzzword *null string* instead of empty word.

There are a few more things worth mentioning about dimension statements. DIM a(15) should mean: set aside 15 slots for the list named *a*. Unfortunately, SmartBASIC interprets this slightly differently. To SmartBASIC, DIM a(15) means set aside 16 slots for this list. The sixteen slots are a(0), a(1), a(2), ... a(15). In other words, BASIC starts all lists with a zero entry. Generally, nobody worries about the extra slot labelled 0 unless they start running out of space in memory or there is something extra they want to add to the list. Starting with zero is a holdover, not only from the original BASIC, but from FORTRAN.

Finally, once you dimension a list you can't easily change your mind. If you try to change the length of the list with a new dimension statement by writing:

```
10 DIM a(23)
     .
     .
     .
1000 DIM a(57)
```

SmartBASIC will respond with:

```
Redimensioned Array Error in 1000
```

and the program will bomb (end).

Tricks of the Trade: There *is* a way to change your mind but only at some cost. The command CLEAR allows you to redimension arrays, but it has many side-effects. CLEAR also sets all variables in your program equal to zero. Using CLEAR in the middle of a program is very similar to another use of the RUN command. ADAM allows you to say:

```
RUN (line number)
```

Instead of starting at the lowest line number in the program, ADAM starts processing at the line number indicated in the command. For example:

```
RUN 100
```

starts your program processing at line 100.

Self-Check 1

A. Questions

1. In our questionnaire, what does *a3(4)* mean?
2. What kind of lists are set up by the following DIM statements?

```
DIM c$(125)        DIM k(12)
```

3. What is wrong with the following program fragment?

```
10 DIM x$(20)
20 LET x$(19)
```

A. Programs

Write modules for the questionnaire that will find:

1. How many people use trains for most of their transportation.
2. How many people are over 60 and bicycle.

3. How many people are either under eleven or usually travel by train or bus.

DATA Statements

Getting data into a computer is not one of life's great joys. In the good old days—say fifteen or twenty years ago—hundreds of people sat in a room, each one facing a large, noisy machine that punched holes in cards. You took these cards and then another noisy machine read these punched cards and sent your program and data into the computer. Having to do this ended about ten years ago—except at most universities. Most universities seem to have outdated equipment. The authors of this book are old enough to remember how miserable it was to punch cards and wait hours or days for their programs to run on the computer—and then find that some silly mistake fouled things up. With microcomputers, results that took days to get fifteen years ago now take seconds. Nonetheless, typing is not our idea of fun and we want faster and easier ways to get data into the computer. Typing INPUT or LET statements is boring and time-consuming. If the data doesn't change much, it's silly to have to reenter it each time a program is run. Using INPUT statements means you'll have to type in the data each time. Lots of LET (assignment) statements make even a simple program that uses many variables incredibly long. Fortunately, there are two other ways to get data into ADAM for use by a program. We'll show you both of these methods, one in this section and the other in chapter 16.

In the previous section we showed you how ADAM filled a list contained in memory by repeated use of an INPUT statement. With this method, data is read from the keys you hit on the keyboard. When data doesn't change much from run to run (RUN is not only a command, it is also a buzzword; it means what happens after you enter RUN), we might as well build the data into the program. The statement that does this is called a DATA statement. To grab information from a DATA list you use the READ command. For example, a program could contain the lines:

```
70 READ a1,a1$(3)
80 FOR i = 1 TO 6
90 READ b(i)
100 NEXT i
```

As soon as ADAM sees a READ statement, it looks for a DATA statement with something to read. It is good practice to use the *highest* line numbers in a program (the lines at the end; or, used with care, the lines at the end of some module) for DATA statements. This makes it easier to find the DATA lines (or statements) if you need to change them. Most programmers use a line or two with empty REM statements to separate the parts of a program as we did in chapter 7. It is a good idea with DATA statements as well. Suppose we added the following line to the program fragment above:

```
200 DATA 5,hi there, 65, -1.4,0,17,23,45
```

We could have used eight separate DATA statements for these items, but using separate statements doesn't take full advantage of the power of DATA statements.

When ADAM sees the first READ statement, it reads the first piece of data on the first DATA statement in the program. If the READ statement says read two items, the first two data entries are read, etc. So line 70 has the same effect as the assignment statements:

```
a1 =5
a1$ = "hi there"
```

The loop on lines 80 to 100 has the same effect as the six assignment statements:

```
b(1) = 65
b(2) = -1.4
b(3) = 0
b(4) = 17
b(5) = 23
b(6) = 45
```

The key point here is that a READ command takes input from DATA statements in order.

In a DATA statement each entry is separated by a comma from the next entry and no comma follows the last entry. If your program says "READ b$," ADAM knows that *b$* will be a string (word) so you don't need to put quotation marks in your DATA statements. There is an exception: if the string contains a comma like GREENBERG, ARNOLD. Here you need the quotes because otherwise ADAM things GREENBERG is one piece of data and ARNOLD is another. If you put in more data than a program calls for by READ commands, ADAM just reads exactly what it needs and never uses the rest. If the first READ statement asks for five items, the second asks for three, and the third asks for ten, the DATA statements must have at least eighteen entries. If you ask for more data than is contained in the DATA statements. You'll get an error message saying:

```
Out of Data in (somewhere)
```

and your program stops running.

Different versions of BASIC usually have only one command to go backwards in DATA lists. The command RESTORE tells ADAM to go back to the first entry in the first DATA list. This command is very useful. For example, one problem with DATA lists is that ADAM must know at least how much data is on the list before reading it. If there is too little data the program will bomb. To prevent this, it is common to have ADAM pass through a DATA list twice. During the first pass, you have ADAM count the number of items it reads until it encounters something strange (like 1E38 or zzzz or −999). Then you have ADAM use dynamic dimensioning to dimension the list with the value of the counting variable. Finally you have ADAM pass through the DATA list a second time to actually read the data. As an example, suppose a program had the following DATA statement:

```
500 DATA 28500,BILL,23540,HARRY,12345,ADAM,
    12345,CHARLES,23456,EVE,87650,,CATHY,
    125000,HELEN,21900,JANE,15000,HARRY,zzzz
```

Rather than you trying to count the number of salaries and people on the list it's easier and less likely to result in mistakes if ADAM does it. Since ADAM can treat a number as a string (of digits), the following trick will work:

```
1000  LET c = 0
1010  READ a$
1020  IF a = "zzzz" THEN 1100
1030  let c = c+1
1040  GOTO 1010
1100  DIM sal(c/2),name$(c/2)
1110  RESTORE
1120  FOR a = 1 TO c/2
1130  READ sal(a)
1140  READ name$(a)
1150  NEXT a
```

The number of data items that ADAM encounters before the delimiter *zzzz* is counted by *c*. (A **delimiter** is a character or combination of characters used to indicate the beginning or end of something else.) After that it's just a matter of going back to the beginning of the DATA list and reading each pair. The test value in the FOR-NEXT loop is *c* / 2 because ADAM has to separate each pair into a salary and a name.

Since the RESTORE command always goes back to the first entry in the first DATA statement, be careful that ADAM is keeping track of the location on the list of the data it wants. Suppose you have ADAM remember (by a trick like the one in the previous program, or some other use of a delimiter) that an item you want is the thirty-fifth piece of data contained in a program. If you needed only that piece of information, ADAM would have to read (and possibly discard) all thirty-four items that preceeded it. The following fragment keeps the thirty-fifth entry and gets rid of all the items that precede it on a DATA list.

```
          .
          .
          .
2000 RESTORE
2010 FOR i = 1 TO 35
2020 READ a$
2030 NEXT i
          .
          .
          .
```

In this fragment, the first through thirty-fourth items in the DATA list are read and then overwritten. When the loop ends, the variable *a$* holds the thirty-fifth entry on the list.

When ADAM is asked to read an entry in a data list it must know exactly what kind of DATA to expect. If you say READ *a* and the next entry on the DATA list is a string, you'll get an error message and the program will bomb.

Self-Check 2

A. Questions

1. What is wrong with the following program fragment?

```
10 READ a,b$,c
20 DATA 30,ex,hello
```

2. In the program on the bottom of page 245, what happens if we reverse lines 1020 and 1030?

B. Program

Write a module to count the number of items on a list until the second of two delimiters is found.

Searching and Sorting

Suppose a long list of names is already entered in ADAM's memory. We want to find out if a certain name is on the list. This can be done easily: just have ADAM compare the name you want with all the names on the list. Since ADAM works very rapidly, this method is usually quite effective.

However, if the list had five thousand names and was already in alphabetical order, this would be a silly waste of time. If you are looking in a telephone book for a name beginning with K you don't start at page 1. You split the book in about the middle and proceed from there. When the information is already ordered you can speed things up by an extension of this method. Each time ADAM looks at a part of the list that is half the size of the previous list. This speeds things up almost beyond belief. Instead of having to look (in the worst possible case) through five thousand names, ADAM will never have to look through more than twelve. Here is an outline for a program to search through a list that is already in alphabetical order.

1. Divide the list in half.

2. Have we gone too far? (Is the entry at the halfway mark before or after the name we are looking for?)

3. If we have gone too far, look at the first half.

4. If not look at the second half.

5. Go back to step 1 using the smaller list.

The outline looks like it should work—but it may not. Suppose the name wasn't on the list. Then none of the steps give us a way out. One way to do this is to have ADAM check if doing step 5 means going back to a list with less than one entry. If ADAM gets to that point we know that the name isn't on the list.

Suppose our list has 5000 names. Then after doing step 3 (or 4) we go back to step 1 with a list of 2500 names. Doing it again we have only 1250 names, a third time only 625, and so on. By the twelfth time there would only be two names to look at. The buzzword for this incredibly fast way of searching is *binary search.* To give you an idea of the speed of this process, imagine you were searching through the New York City telephone directories (they have about 10,000,000 numbers) and you had to find out whether someone's name is there. Just following this outline (and not doing any estimating of where the letters are) you could find out if the name is there in less than twenty-five applications of the fifth step.

This program is a bit tricky. But it's worth spending some time on.

```
 997 REM ************************************************
 998 REM  binary search module
 999 REM ************************************************
1000 PRINT "This program looks for a number in an ordered
list"
1010 INPUT "What number are we searching for?"; t
1020 INPUT "How long is the list? "; l
1030 LET b = 1: LET e = l: REM l is the length of the list
1040 IF e - b <= 1 THEN 1200
1050 LET q = (b+e)/2
1060 LET h = INT(q+.5): REM round off q
1070 IF x(h) < t THEN b = h: GOTO 1040
1080 IF x(h) > t THEN e = h: GOTO 1040
1099 REM *******CHECK IF DONE*********************************
1200 IF x(b) = t THEN PRINT "Your number was number "; b; " on
the list" : GOTO 1300
1210 IF x(h) = t THEN PRINT "Your number was number "; h; " on
the list": GOTO 1300
1220 IF x(n) = t THEN PRINT "Your number was "; e; " on the
list": GOTO 1300
1230 PRINT "Your number isn't on the list"
1300 STOP
```

This module is short but it is a bit harder than it looks. In the outline we said we have to divide the list into halves. We decided to use *b* as the variable marking the beginning of each half and to use *e* for the variable marking the end of the half. (These marking variables are called *markers.*) Notice we initialize *b* to be 1 and *e* to be 1 (the length of the list).

First, ADAM checks if the beginning of the list and the end are at most one apart. This line is important—it gives ADAM a way to check if the search is complete (by going to the sub-module on lines 1200 and 1230). Otherwise ADAM finds the halfway mark by averaging *b* and *e.* Since this won't always be a whole number, we round it off using the technique from the last chapter. This new variable *h* marks the halfway point. If *x(h)* is smaller than the target *t,* then whatever was at the halfway point was too small. That means we must change the halfway point into the new beginning point. If what is at the halfway point is bigger than the target, ADAM makes the halfway marker the end marker. Notice that if *x(h)* is equal to *t,* ADAM falls into line 1200. (Lines 1070 and 1080 have no effect because their IF clauses are false.)

Line 1300 is a new command. The command STOP acts like CONTROL/C. When ADAM encounters this command, it stops processing and waits for you to give it further instructions. As with CONTROL/C, the command CONT starts processing again. We used it here to mark the end of the module. After writing a module, you should test it before including it in a long

program. Testing parts of a program is a real art. In Interlude 2, we will show you some ways to test parts of a program.

People like ordered lists such as dictionaries, telephone books, and numbered pages in books. Computers are fond of them, too, because techniques like binary search work so quickly. Rearranging data so that it is ordered is one of the most common demands placed on a computer. You probably have spent a considerable amount of time rearranging things into alphabetical order as well. Alphabetizing isn't much fun, so it would be nice to have ADAM take over the drudgery. Unfortunately, sorting is also one of the most time-consuming of the tasks a computer can do. In this section we will show you a method that works quite well for lists with up to a few hundred entries. For lists that are much longer you'll have to consult a more advanced book on programming to find more sophisticated programs and techniques.

Writing a sorting program isn't like writing any of the other programs we've already shown you. It's not only a question of analyzing the problem and outlining the steps; it's not even clear how to begin.

Although it may not always work, a good way to start analyzing a hard problem is to try to solve an easier (but related) problem. We already showed you (in chapter 5) how to find the smallest of three numbers. Here's the program again:

```
10 REM this program lets you find the smallest of three
numbers
20 INPUT "Your three numbers please "; a,b,c
30 LET s=a: REM we'll try a as the smallest
40 IF b<s THEN LET s=b
50 IF c<s THEN LET s=c
60 PRINT "The smallest number was "; s
70 END
```

ADAM started with the temporary variable equal to the first variable, then it compared its value with the other two variables. Whenever one of the other variables was smaller it replaced the temporary variable. It's easy to modify this program, using lists, to find the smallest of ten, twenty, or any number of entries. We will rearrange it so that the smallest entry (or first in alphabetical order) becomes the first entry. To reorder the numbers, we will need a temporary variable to allow the switch.

```
1997 REM ******************************************************
1998 REM   Finding the smallest number in a list
1999 REM ******************************************************
```

(continued)

```
2000 FOR i = 2 TO n: REM n is the length of the list
2010 IF x(i) > = x(1) THEN 2050
2020 LET t = x(1)
2030 LET x(1) = x(i)
2040 LET x(i) = t
2050 NEXT i
2060 PRINT  x(1); " is the smallest entry"
2070 END
```

If ADAM now finds the smallest entry on the remaining list (from 2 up to *n*) then the first two entries will be in the right order. By having ADAM continue in this way, eventually the list will be in the correct order (alphabetic order or smallest first, next-smallest next, and so on). Here is a program module that continues to shrink the size of the list after pulling out the smallest number:

```
1997 REM ***************************************************
1998 REM  a sorting module
1999 REM ***************************************************
2000 FOR j = 1 TO n-1
2010 FOR i = j + 1 to n
2020 IF x(i) >= x(j) THEN 2060
2030 LET t = x(j)
2040 LET x(j) = x(i)
2050 LET x(i) = t
2060 NEXT i
2070 NEXT j
2071 REM ******END SORT*************************************
```

Lines 2010–2060 are the same as in the previous module. The only difference in this module is that to completely order the list we want ADAM to repeat the same process, with the list shrinking each time. On the first pass through the loop, *j* is 1, so *i* goes from 2 to the end of the list. On the second pass, *j* is 2, and *i* goes from 3 to the end of the list, and so on. On each successive pass, ADAM works with a smaller list until it finishes. After the last pass the entries in the list are arranged so that each entry in the new list is no smaller than the previous entry.

(If you want to alphabetize lists, just change the numeric variables to string variables. Remember that any capital letter comes before any lowercase letter.)

This method of sorting is usually called an *exchange* or *ripple sort.* It works quite well for small lists. If you enter it and ask ADAM to sort a list of two hundred names it takes about 2½ minutes. Unfortunately, a list of four hundred names takes about 10 minutes. This kind of sort has the unfortunate property that doubling the length of the list quadruples the time. This means that a list of eight hundred names takes about 45 minutes. If you have lists this long to be sorted you should refer to a more advanced programming book. A pair of our favorites are the books *Subroutine Sandwich* and *More Subroutine Sandwich* by John P. Grillo and J. D. Robertson (both published by the Wiley Press). These books not only present some excellent sorting programs, but the programs are also well-written; they can serve as models for good programming style. (These books use a slightly different version of BASIC. At this point you should not have any trouble translating these programs into SmartBASIC.)

Self-Check 3

A. Questions

1. On line 2000 of the program on the bottom of page 250; why is the test value *n-1* instead of *n*?
2. On line 2010 of the same program, why is the starting value *j+1*, not *j*?
3. What changes do you have to make so that the program arranges lists so that the largest number is first and the smallest last?

B. Programs

1. Write a program that uses the SORT module to alphabetize a list that someone enters. You will have to dimension and input the list before sorting it.
2. Write a program that generates a list of one hundred random numbers and then sorts these numbers using the SORT module.

Summary

In this chapter, we showed you how to use lists. Each entry on a list is named by the name of the list and its position in the list. The position is called the subscript. You learned how to dimension a list using numeric expressions. We showed you different techniques to search through lists for particular numbers or strings and how to sort the entries in a list. You learned the following commands:

DIM	sets aside space in ADAM's memory for the entries in a list.
DATA	is a statement in a program containing a list of numbers and/or strings.
RESTORE	restarts the reading of the DATA list at the first entry.
STOP	halts the execution of a program.
CLEAR	reinitializes all numeric variables to zero and all string variables to the null string. It allows us to redimension lists while a program is running.
RUN (line number)	starts the execution of a program at the specified line.

BUZZwords

BINARY SEARCH

DEFAULT VALUE

DELIMETER

DYNAMIC DIMENSIONING

EMPTY WORD

EXCHANGE SORT

INITIALIZE

LIST

NULL STRING

RIPPLE SORT

RUN

SORT

SUBSCRIPT



Arrays

In this chapter we will show you *arrays,* which are another way of organizing variables. Instead of having one subscript like a list, an array can have two, three, or more. We'll also show you a new type of variable called the *integer variable.* Integer variables can help some of our earlier programs to run faster and be easier to use. Integer arrays also save space in memory. You'll learn how to check how much memory is available for your program and variables, and about relative coordinates and scaling. The only new command you'll see is:

FRE(0)

Array Variables

This section is about variables with more than one subscript. These are called *arrays.* Just as variables with one subscript come from lists and are used every day, variables with two subscripts are also familiar. They come from tables (of data).

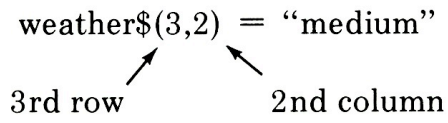
Suppose you were looking at a table of the average weather conditions in five cities in each of the four seasons. It might look like table 11.1.

City	Summer	Fall	Winter	Spring
New York	bad (2)	good (7)	tolerable (4)	good (8)
San Francisco	good (7)	good (8)	medium (6)	great (10)
London	good (7)	medium (5)	poor (3)	good (7)
Paris	medium (5)	good (7)	tolerable (4)	good (8)
Stockholm	great (10)	poor (3)	horrible (0)	good (7)

Table 11.1

This table is written in two different ways—with words and with numbers on a scale from zero to ten. For the moment we will work with the words. This table *could* be stored in ADAM using six string lists: season$, city$, summer-weather$, fall-weather$, etc. However, we rarely think of a table as being made up of separate lists, so we shouldn't store it as lists in ADAM's memory. Besides, information is organized into tables because both the rows and columns provide information.

This particular table is made up of five rows and four columns and might be named *weather$*. The entries in this table are simply the words: good, poor, medium, etc. Suppose we look at the entry in the third row and second column—it's the word *medium*. Because it lies in the third row, it's about the weather in London. Since it is also in the second column, it describes the weather conditions in the fall. These three pieces of information tell us that London's weather in the fall is medium. There is a fairly standard notation for describing entries in tables. A picture helps make the notation clear.



So weather$(3,4) means the entry in row 3 and column 4 and the *$* tells us that the array uses words (strings). The position of this entry in the table shows that it describes the spring weather in London and the entry in that position says that the weather is good.

Unfortunately, ADAM's lo-res graphics screen works in the opposite way. There *the first number is the column and the second is the row.* It is necessary but annoying to remember which convention is being used—rows first or columns first.

To a computer, a table is a collection of variables, similar to the variables on a list but having two subscripts instead of one. An *array* is the computerese word for a table. Usually the variables *i* and *j* are used for the subscripts—*i* is the row and *j* is the column. The *i* is supposed to remind you of the word index—subscripts keep track of or *index* the entries. The *j* is used because it follows *i* in the alphabet.

In all versions of BASIC, arrays can be made up of numbers or words. An array whose entries are strings (words) is called a *string array.* If the entries are numbers it is called a *numeric array.* Just as with lists, arrays with ten or fewer rows and columns don't have to be dimensioned. But all others must be dimensioned. Because arrays use up space much faster than lists, it is a good idea to dimension all arrays. The DIMension statements to use for the weather arrays are:

```
DIM weather(5,4)
```

or:

```
DIM weather$(5,4)
```

The first statement is used for the numeric data and the second for the string data. As before, only the first two characters in the (array) variable name are recognized by ADAM. To avoid having to type the word weather each time, we'll call the arrays *we* and *we$.*

Entering data into an array that is in ADAM's memory can be done using INPUT, READ, and LET statements. READ is by far the easiest to use. Here's an example:

```
10 REM to input the weather table
20 DIM we$(5,4)
30 FOR i = 1 TO 5
40 FOR j = 1 TO 4
50 READ we$(i,j)
60 NEXT j: NEXT i
        .
        .
        .
2000 DATA bad, good, tolerable,good
2010 DATA good, good, medium, great
2020 DATA good, medium, poor, good
2030 DATA medium, good, tolerable, great
2040 DATA great, poor, horrid, good
```

The DATA statements are arranged in exactly the same form as the table. Each DATA statement (or line) corresponds to a row in the table. This demonstrates just one of the advantages of using DATA statements to enter information into arrays. ADAM will read and store all the columns (the *j*'s) in the first row (i = 1) before beginning to read the columns in the second row (i = 2), and so on.

For most tables, you'll want to have a list of the columns and row headings. For this table, these can be read from DATA lists by adding the following lines to program above:

```
70 DIM cty$(5),sea$(4)
80 FOR i = 1 TO 5
90 READ cty$(i)
100 NEXT i
110 FOR j = 1 TO 4
120 READ sea$(j)
130 NEXT j
2050 DATA New York, San Francisco, London, Paris,
Stockholm
2060 DATA summer, fall, winter, spring
```

Now all the information contained in the table is in ADAM's memory. Unfortunately, we have used words as entries in the table and a computer can't do much with words. For example, ADAM can't average *good* and *bad* - we can't either. Numbers can be averaged and compared directly. This is why most questionnaires, test scores, and other information fed into computers use numbers.

Displaying arrays isn't easy, especially if the arrays are string arrays. On a television screen, it is next to impossible to display an array meaningfully.

The screen will almost always be too small to display the entries in their correct columns and rows. An alternative is to try the printer. If there aren't too many columns and the entries are short, the printer is adequate. However, even here, printing arrays so that they look like tables requires locating and lining up decimal points and finding the length of words. We'll show you how to do this kind of formatting in chapter 13.

Here's a simple way to print the weather array (which is small enough to do without using complicated techniques).

```
140 PR#1: TAB(1)
150 FOR j = 1 TO 4
160 PRINT sea$(j),
170 NEXT j
180 PRINT
190 FOR i = 1 TO 5
200 PRINT: PRINT cty$(i),
210 FOR j = 1 TO 4
220 PRINT we$(i,j),
230 NEXT j: NEXT i
240 END
```

The best way to see what we mean by the problems of displaying arrays on the television is to watch the screen as this program feeds the array to the printer. The display on the screen doesn't look anything like a table. Columns float all over the screen and nothing is aligned (*aligned* is the buzzword for lined up) as it should be in a table.

ADAM can also build arrays the same way it builds lists. Here's a short program that sets up a table of powers of numbers from two to six. The powers range from one to ten.

```
10 REM powers of some numbers - an example of an array
20 DIM a(10,5)
30 FOR i = 1 TO 10
40 FOR j = 1 TO 5
50 a(i,j) = (j+1)^i
60 NEXT j: NEXT i
```

If you want a copy of this table, you can modify the print module (program at the top of this page) to do the printing.

Arrays are not limited to having one or two subscripts; ADAM lets you

use as many as you could ever want. (We've never used more than four subscripts in any program, no matter how long and complicated.) When you use arrays, there are several things to keep in mind. The DIMension statement must show how many subscripts you are using and the data read must fit into the array set up by the DIMension statement. Finally, each subscript in the DIMension statement must be at least as large as the subscripts used in the Program. If you say "DIM*a*(3,13)" and then ask ADAM to fill position *a(2,14)* (that is, row 2, column 14 in the array *a*), you'll get an error message.

If your arrays have several subscripts, ADAM's memory runs out fast. If you were keeping records of rainfall in your area then you would probably use an array variable with three subscripts: *rainfall*(*i,j,k*). The *k* could stand for the year (take the first year that you are interested in as year 0), the *j* for the month, and the *i* for the day of the month. The dimension statement is DIM *rainfall* (31,12,★). The star stands for the number of years of rainfall data that you want to have in memory. About twelve years is all that will fit. (If you think that twelve years isn't a lot of data, you might want to think about how long it would take you to enter all of it. The number of entries in an array is the product of the subscripts given in the dimension statement. A table with twelve years of rainfall data has about 31*12*12, or 4464 entries.)

As with lists, the size of an array can be set using variables. This allows a program to compute the size of an array that it will use. Then the program can dynamically dimension that array. An array, like a list, can only be redimensioned after a CLEAR command and this command has many side effects (see chapter 10).

One example of a variable in a dimension statement is:

```
200 a = 5*b + 1
800 DIM f(a,16)
```

which makes *f* an array with *a* rows and 16 columns. Another example is:

```
10 PRINT "How many rows and columns do you
   want the array to have?"
```

```
20 INPUT "Enter the numbers separated by a
   comma ";r,c
30 DIM q(r,c)
```

Self-Check 1

A. Questions

1. In our weather table what does *we(4,2)* stand for?
2. How many entries does DIM *s*(3,3,2) allow?
3. Why won't the following program run?

```
10 DIM a(23,11)
20 FOR i = 1 TO 35
30 FOR j = 1 TO 10
40 a(i,j) = i+j
50 NEXT i:  NEXT j
60 END
```

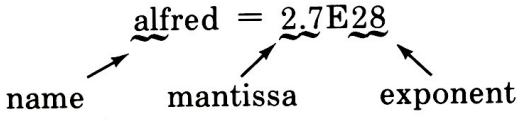
B. Programs

1. Write a program that determines the city with the best weather by using the array *we* and averaging the weather in each city.
2. Write a program that reads the monthly rainfall data for four cities into an array. Write separate modules that compute the average monthly rainfall and total annual rainfall in each of the cities.
3. In chapter 9 we showed you how to dimension a list by making two passes through the DATA statements. You have to be a little more careful when you do this with tables. Change the weather program so that the number of cities being compared is computed from the DATA statements. (You'll have to insert terminators into the data lists.)

Integer Variables and the Use of Space in Memory

With most programs in SmartBASIC, it isn't necessary to worry about having enough room in memory for the program and the variables used. As we explained in chapter 10, string lists can "eat up" memory quickly. A string list of ninety to one hundred entries can use up all of ADAM's memory and you still need room for the program. Arrays eat up memory even faster. In this section, we will explain how different variable types use space in memory. (*Variable type* is the buzzword for a variable like a string or numeric variable.) We'll also show you another variable type, the integer variable, which helps to conserve space. This kind of variable has some peculiarities; we will explain some of them here as well.

So far we have only used one type of numeric variable and haven't even told you its usual name. These numeric variables have decimal points and are called *real variables* or, more traditionally, *floating point variables.* They are stored in ADAM in scientific notation. The largest number that ADAM can handle is about 1.7E38 and the smallest positive number is 0.00000001E-38. Numbers can also have positive or negative signs. In chapter 2 we said that ADAM's memory contains a large number of memory locations, each one containing eight bits or one byte of information. This is roughly one character. Each real variable uses up ten bytes of memory. ADAM needs two bytes for the variable's name, 2 for its location in memory, 4 bytes for the *mantissa* (the number before the E), and 1 byte for the exponent. (This doesn't account for one byte. That byte has no function as far as using SmartBASIC is concerned. ADAM uses it to keep its house in order.) Suppose:



A string variable takes at least five bytes and often much more because it needs one byte for each character in the string.

There is one other variable type. It exists for at least two reasons: first, it

is traditional (which means it exists in FORTRAN) and, second, it saves space. The variable type is called the *integer variable* type. The value of an integer variable must be an integer (whole number) between −32767 and +32767. If you try to give an integer variable a value not satisfying these requirements the program will bomb. Integer variables save space—they use only five bytes of storage. When you use arrays, you get even larger savings. Just as we needed an identifier, the dollar sign, at the end of a string variable, integer variables have a special identifier as the last symbol in their names. (*Identifier* is the buzzword for a special symbol used as a label.) The integer variable identifier is the percent sign, %. So *a%* is the name of an integer variable. To ADAM, *a*, *a$*, and *a%* are three different variable names and you can use them all in the same program.

Integer variables have a slightly peculiar arithmetic. Whenever ADAM divides integer variables it will assume their values are decimals. If:

$$a\% = 6 \qquad \text{and} \qquad b\% = 4$$

then in computing:

$$a\% \,/\, b\%$$

ADAM temporarily converts both numbers into real variables and then does the division. If you enter these values for *a%* and *b%* and then the command:

```
PRINT a%/b%
```

ADAM will respond with the number 1.5. In contrast, if we enter:

```
c% = a%/b%:  PRINT c%
```

ADAM will display the number 1. This really isn't that strange. The percent sign acts as if it were the function INT (the greatest integer function from chapter 9).

Integer variables have one other minor advantage besides the space

they save. Integer addition and subtraction is performed extremely fast. Since ADAM is fast—even by microcomputer standards—this isn't very important.

You can have integer arrays as well. The term *a%(3,2)* is the entry in the third row and second column in an integer array. Integer arrays obey the same rules as real arrays and string arrays do.

One way to see the space savings that integer arrays give is to use another SmartBASIC command, called FRE*(0)*. This command will tell you how much memory, measured in bytes, ADAM has available for your program. (The zero is a placeholder; it has no meaning but must be included.) For example right after the command NEW, all of ADAM's memory is available to store programs and variables. If the next command you enter is:

```
PRINT FRE(0)
```

ADAM responds with the number 25954. (This number may be slightly different in different versions of SmartBASIC.)

This command is useful in programs that use most of the available memory. You can check to see whether ADAM is about to run out of memory. If ADAM runs out of memory, a program bombs. (Imagine entering twenty-five pages of data and then having the program bomb so that you have to start all over again. Computers have been thrown out of windows for less.) In programs using most of memory, you will often find statements like:

```
IF FRE(0) < 100 THEN PRINT "HELP I"M ALMOST
OUT OF SPACE - GET ME A 64K EXPANDER": GOTO
9500
```

Line 9500 might be the start of a module that saves as much data as possible in a file (see chapter 16) before the program bombs. The check of free space is an example of an *error trap* and the module at line 9500 is a *data-saving module.*

You can also use the command FRE*(0)* to see the difference in how the

different variable types use memory. Try the following three programs (remember to enter NEW before each program).

First to see how much space real arrays use, run the following program:

```
10 DIM a(50,50)
20 PRINT FRE(0)
30 END
```

Next, try the string array:

```
10 DIM a$ (50,50)
20 PRINT FRE(0)
30 END
```

Then try the integer array:

```
10 DIM a%(50,50)
20 PRINT FRE(0)
30 END
```

Finally:

```
10 DIM a(2500)
20 PRINT FRE(0)
30 END
```

Doing this shows that integer and string arrays use less space in memory than the same number of variables in a list. Twenty-five hundred individual variables would take up even more space. (If you compute the average number of bytes used by an entry in these arrays, you'll find that integers and strings use about 2.1 while the real array entries use about 9.9 bytes.) The explanation is simple. For each variable, ADAM has to store the name

of the variable and the memory location where ADAM starts storing the variable. Each of these uses two bytes of storage. When ADAM stores integer and string arrays, it stores only one name for the array. This saves a lot of space, especially for integer arrays. (Remember that the string array entries were blank. As soon as you fill them, string arrays take up much more space.)

In working with ADAM's low-res graphics screen, the codes for the colors that we might want to paint at a particular block are integers. The position of each block is given by its row and column. So we can represent a painted low-res screen as an integer array. (*Represent* is sort of a buzzword meaning "think of.") This array has forty rows and forty columns, each numbered from 0 to 39. A piece of paper set up for typewriter art can also be represented by a sixty-five-row, eighty-column array. Even though you print characters, it is a good idea to use integer arrays even for typewriter art because these arrays use a lot of space—sometimes much more than you can spare.

Here's a different type of *paint-brush program.* This time we will transfer the information directly from a grid to ADAM's memory. Then we will have ADAM paint the whole picture at once. When the painted blocks are far apart, this program is more efficient than the original paint-brush program (see chapter 7, page 266).

Here's the outline and the program:

1. Start with a 40x40 array filled with zeroes. (Zero is the color code for black.)

2. Have the painter enter the position of the square to be painted.

3. Change the entry at the position (from step 2) in the array from 0 (black) to whatever color is wanted.

4. Repeat steps 2 and 3 as often as needed.

5. Paint the design on demand.

```
10 REM a new paint-brush program
20 HOME
30 DIM a%(39,39)
40 PRINT "This program will let you transfer a design directly
from graph paper to ADAM's screen"
```

(continued)

```
50 PRINT "Enter s to start"
60 INPUT s$
70 IF s$ = "s" OR s$ = "S" THEN 500
80 TEXT
90 END
495 REM
496 REM
497 REM ******************************************************
498 REM  INPUT MODULE
499 REM ******************************************************
500 PRINT "Do you want to see what you've painted?"
510 INPUT "Enter yes or hit return to indicate no. ";yn$
520 IF yn$ = "yes" THEN 1000
530 HOME
540 PRINT "Enter the column and the row sparated by a comma."
550 INPUT c,r: HOME
560 PRINT "The position is column ";c;" and row ";r
570 INPUT "What color do you want (enter the code)? ";d
580 IF d < 0 OR d > 15 THEN 600.
590 a%(c,r) = d: GOTO 500
600 HOME
610 PRINT CHR$(7);"Your color code isn't allowed"
620 INPUT "Enter q to quit or hit return ";q$
630 IF q$ = "q" THEN 80
640 GOTO 500
995 REM
996 REM
997 REM ******************************************************
998 REM       PAINTING MODULE
999 REM ******************************************************
1000 GR
1010 FOR col = 0 TO 39
1020 FOR row = 0 TO 39
1030 COLOR = a%(col,row)
1040 PLOT col,row
1050 NEXT row
1060 NEXT col
1070 HOME
1080 PRINT "Do you want to quit"
1090 INPUT "Enter q to quit ";q$
1100 IF q$ = "q" THEN 80
1110 GOTO 530
```

Notice that we dimensioned the integer array at 39,39—this means that we are using the entry zero in each row and column. This is more convenient because the low-res graphics screen has row and columns numbered from 0 to 39. We can use the same subscripts for the array as for the rows and columns. The input module lets you color each square by entering a code into the integer array. After the codes have been entered to the painter's satisfaction, the painting module takes over. This module checks that the

color codes for each block are allowed (so that the program doesn't bomb) and draws the design. If the color code isn't allowed, we have ADAM ring its bell (see chapter 8) and print a message.

Self-Check 2

A. Questions

1. What kinds of variables are *able, able$,* and *able%*?
2. In the paintbrush program, suppose *a%*(20,20) = 14. What would ADAM do with this information in the paint module?

B. Programs

1. Add a module to the new paint-brush program that changes the background color.
2. Write an input module for a typewriter art program. (You have to set up a code table if you want to use integer arrays.)
3. Can you see how to combine the two paint-brush programs into one program?
4. Write a program to test the speed advantage of integer addition and subtraction over the same operations with real variables. (Hint: Use two separate programs, one for the real array and one for the integer. In each, set up a twenty-row, twenty-column array. Add 1 to an entry a thousand times, then go to the next entry. Time the run with a watch.)

A Graphing Program

This section will be most useful for those who need graphs in school or work.

If some of the terms seem strange—perhaps too mathematical—there is nothing wrong with skipping this section, although the same methods are

used in many types of hi-res graphics programs. A printer like the SmartWriter can print graphs of functions in two different ways. We'll show you one way here. For the other, see chapter 15.

When you graph any function the first step is to draw the X and Y axes. Usually the X axis is horizontal and the Y axis is vertical. The place where they meet is called the *origin.* Next a scale is marked on each axis. (This step is easier if you use graph paper). The scale lets you determine (approximately) the location of every point. For example, suppose you were looking for the point (3,4). The *3* is the X coordinate and, because it is equal to 3, you should move over to the right three units. The *4* is the Y coordinate and, because it is 4, you should move up four units. In general, if the X coordinate is positive you move to the right of the origin, and if it's negative move to the left. The Y coordinate determines whether to move up or down. Move up if the Y coordinate is positive, move down if it's negative.

Suppose for example, your teacher asked you to graph:

$$y = 2x - 1 \qquad \text{(in computerese, y = 2*x - 1)}$$

A graph is a picture of the values of the function. The first step in graphing any function is to make up a table listing some different X values and their corresponding Y values. In our example, all we have to do is substitute selected X values into the equation and find the associated Y values. The table might look like:

x	y
3	5
2	3
1	1
0	−1
−1	−3
−2	−3
−3	−7

Table 11.2

Each of these pairs determines a point on your graph paper. Just as we found the point *(3,4),* it is easy to locate these points. After finding them you plot (draw) a ●. When you've plotted all these pairs you have a rough

picture of the line. (If the equation were more complicated than a simple straight line you would need to find lots more pairs before the rough picture gives you an idea what the graph looks like.)

Let's stop now and imagine how to mimic on a piece of paper what we have just done by hand on our graph paper. Imagine the X and Y axes running through the center of the paper. The center of the paper corresponds to the origin. Suppose we wanted to find a place on the paper that corresponds to, say, the point *(3,5)*. The most natural thing to do is to start with the printhead at the center of the paper. Next, space over to the right one unit and move the paper up 10 units. (Of course this part would have to be done by hand). Now have ADAM print a * at this point. Suppose we had ADAM do this kind of printing for each pair of points in the table. After doing this the paper would also contain a sketch that mimicked the picture of the line.

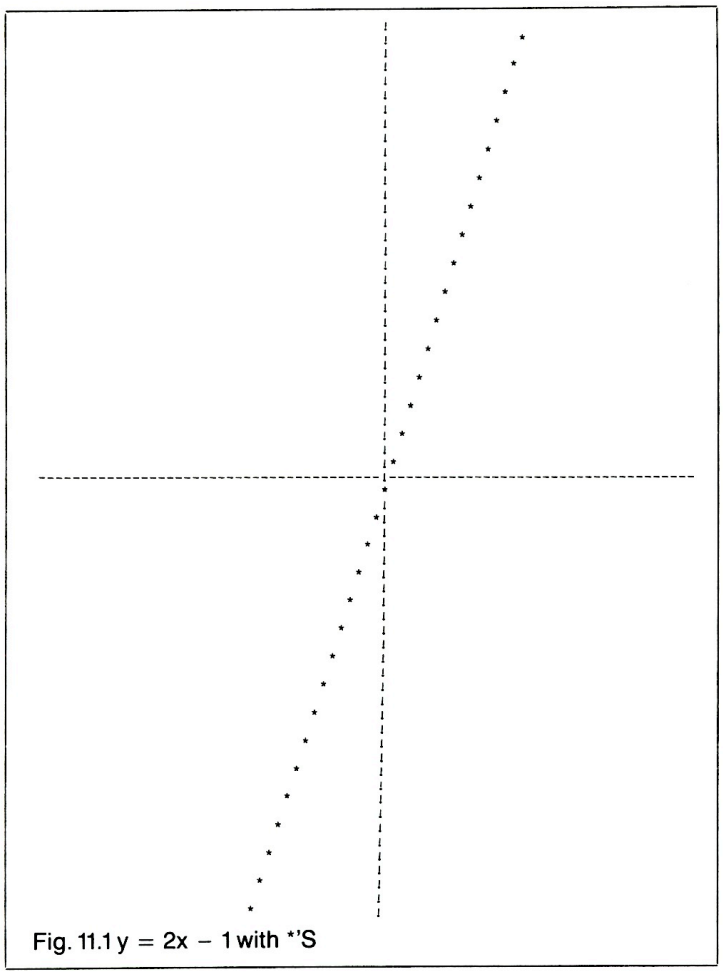


Fig. 11.1 y = 2x – 1 with *'S

You can see that this gives a reasonable first impression of the line. Putting more points in our table and then printing more *'s makes this drawing even better.

The problem with this method is that the SmartWriter can't reverse a carriage return. This means we have to move the paper around quite a lot by hand. We need a trick to print all the *s that belong in a row before moving on. Using an integer array can help. We set up an array and use it much like the array we used in the new paintbrush program (see page 266). This time we will set up an 80x65 integer array and put a *marker* wherever we want the SmartWriter to print a *. Unfortunately, arrays can't have negative indices, so we have to twist a little. One way to do this is to use the middle entry in the 80x65 array as the origin. Then for any other points you move around in the array from this middle point. Look at the following figure:

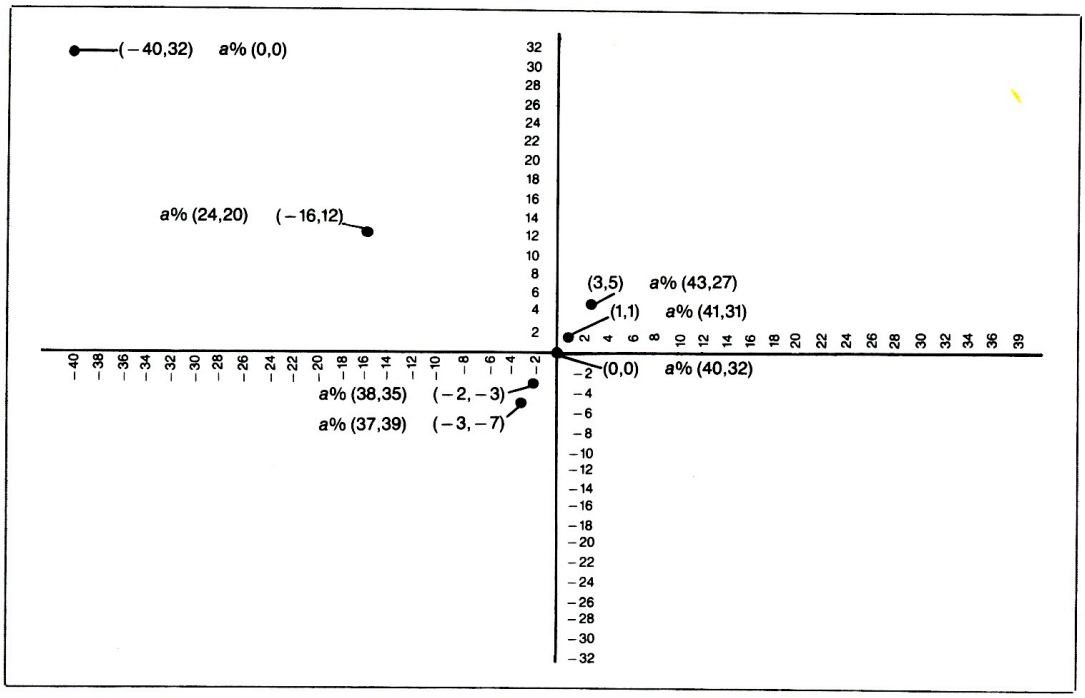


Fig. 11.2 Correspondence between points on a grid and array entries.

In general, to find the array element corresponding to an X-Y coordinate:

add 40 to the X coordinate

subtract the Y coordinate from 32

To go backwards from an array element to an X-Y coordinate, do the opposite:

$a\%(66,22) \longleftrightarrow (26,10)$
$a\%(20,10) \longleftrightarrow (-30,22)$

The subscripts in the integer array are the *relative coordinates* of the points. This buzzword just means we are using different values of X and Y as the origin—(40,32) in this case. Changing the location of the origin is necessary because arrays are not allowed to have negative subscripts.

Now to the actual graphing: we place one kind of marker in the array entries when we want ADAM to print a * and another kind of marker if we want ADAM to just space over. For example:

X-Y coordinate	marker at
Since (1,1) is on the line	$a\%(41,31) = 1$
Since (2,4) isn't on the line !	$a\%(42,28) = 0$

and so on

Very few changes are necessary to outline a general graphing program. We want to do the following:

1. Convert a column index to the corresponding X value.
2. Find the Y value corresponding to this X value.
3. Convert this Y value to its relative coordinates in the integer array.
4. If this array element isn't outside the range of permissible subscripts place the marker *1* in the corresponding array element.
5. Proceed to the next column value.

At the end of these steps the array will have a few 1s in certain places as markers, but most of the array entries are 0.

Now to print the graph:

6. Move column by column along a row of the array and have ADAM print a * if the array element is a 1 or a space if the array element is a 0.

7. Because the printer automatically returns the carriage after it moves through eighty positions on a line, we tell ADAM to move to the next row in the array.

8. We repeat steps 6 and 7 until the entire array has been used.

Finally we might as well print the axes. We'll print :s for the Y axis and —s for the X axis. To do this, we'll use special markers—the numbers 2 and 3—in the positions in the array that represent the axes. By first initializing the array elements that represent the axes before doing steps 1–5, we can change the marker if we want ADAM to print a * at a point on an axis.

The following program works for any straight line.

```
10 REM typewriter graphing program
20 INPUT "Enter m and b for your line ";m,b
30 DEF FN l(x) = m*x + b
40 DIM a%(79,65)
50 FOR q = 0 TO 79
60 a%(q,32) = 3
70 NEXT q
80 FOR q = 0 TO 65
90 a%(40,q) = 2
100 NEXT
496 REM
497 REM
497 REM ************************************
498 REM       FILL ARRAY
499 REM ************************************
500 FOR q = 0 TO 79
510 xval = q - 40
520 yval = FN l(xval)
530 IF ABS(yval) < = 32 THEN a%(q,33 - yval) = 1
540 NEXT
997 REM ************************************
998 REM        PRINTING MODULE
999 REM ************************************
1000 HOME:VTAB(12):FLASH
1010 PRINT "Please insert a piece of paper in the printer"
1020 INPUT "Hit any key when ready."; a$
1030 NORMAL
1040 PR#1: PRINT: HOME
1050 FOR w = 0 TO 65
1060 FOR q = 0 TO 79
1070 IF a%(q,w) = 0 THEN PRINT " ";
1080 IF a%(q,w) = 1 THEN PRINT "*";
1090 IF a%(q,w) = 2 THEN PRINT "|";
```

(continued)

```
1100 IF a%(q,w) = 3 THEN PRINT "-";
1110 NEXT: NEXT
1120 PR#0
1130 END
```

This program works reasonably well but it can be substantially improved. For one thing because standard paper is 8½ x 11 the lines won't quite be in proportion but a little scaling (see the next section) will fix this. We won't worry about that yet.

A more serious problem is that to graph anything other then simple straight lines, you can't assume the Y value corresponding to an X value is always an integer. So to write a more powerful program you have to replace step 2 by:

2+ Find the Y value corresponding to this X value and round it off.

Trick of the Trade This program could be even further improved by not assuming that each array element corresponded to a *grid-point.* (A grid- or lattice-point is the word some teachers use for points that lie directly above or across from integer points on the axes). For example you could make each column and row correspond to a half-unit. This changes the values of x that can be used—now they must be between -20 and $+19$. After you change the scale, you couldn't just round-off the Y values (because each time the y variable changes by one-half, the * changes row). Instead, you have to see which row value in the array they have. After reading the next section you might want to change the program to allow re-scaling.

Self-Check 3

A. Questions

1. If the origin is located at (140,90) and each number stands for one unit, what are the relative coordinates of the following points?

   a. (10,6) b. (24,86) c. (10,108)

2. If the origin is located at the same point as above, suppose some points had relative coordinates:

   a. (160,98) b. (120,120) c. (100,20)

What would the usual (X-Y) coordinates be?

B. Programs

1. Write a menu module that would let the operator choose one function from a list and have that function graphed.

2. Modify the graphing program so that you can print more than one function at a time. (You'll have to prepare two arrays and then combine them. Why not see how much memory is left?)

3. Another possibility that wouldn't use up the space for two arrays is to allow the person to reinsert the paper and then change the function. Make the necessary changes to allow this.

Scaling

The program to draw bar graphs in chapter 7 had a problem. If the data hadn't been adjusted to be at most thirty-eight units, the graph wouldn't fit on the screen. The program just couldn't handle bars whose height was bigger than thirty-eight units. If one of the bars needed to be fifty units high and ADAM was asked to plot it, ADAM would have sent out an error message. We said then that it isn't hard to have ADAM re-scale the data so that it fits on the screen. We want to do that (and more) in this section.

Scaling is something that is quite common, though we don't often make a fuss about it. Scale models of airplanes and cars can be bought at many stores; dollhouses and dollhouse furniture are also scale models. A globe is a scale model of the earth. A map is also scaled. Many maps tell you the scale that has been used. For example, one inch on a roadmap might mean five miles. (We used scaling in chapter 9 to obtain random numbers whose values were not always between zero and one).

Any scale model has all of its parts reduced (or sometimes, increased) in size by the same factor. On the map we just mentioned, a twenty-mile distance shows up as four inches since a 20-mile length is just four 5-mile

lengths. The scale requires dividing the mileage by 5 to find the number of inches. So if you had a 115-mile road, the road would show up as a 23-inch line. To rescale data, you have to do exactly the same thing. You multiply every number by the same amount. This amount is called the *scale factor.*

Now back to the bar graph. Before you can decide what the program should say, you have to work through a few examples. Always start with an easy one. So suppose there are 36, 60, and 76 units to be represented by bars. Seventy-six units can be squeezed down so that they fit in a 38-unit space by dividing by 2. (Stop for a second and think how you figured that out. We told ourselves that 76 is twice as big as 38, so it is twice as big as it can be to still fit in the available space.) Now all the numbers have to be divided by 2 to keep them in proportion. Dividing by 2 is the same as multiplying by 1/2. This is exactly the number you get when dividing 38 by 76. So the scale factor in this case is 1/2 (= 38/76).

Suppose we had to represent 85 units instead of 76. We reason the same way. We have to multiply everything by 38/85 to make the largest bar fit into our screen. All the other numbers also must be multiplied by the same scale factor or else the relative sizes of the bars will no longer be the same (they would be "out of scale"). Without rescaling all the bars, the 36-unit bar would look bigger than the 85-unit one.

Now that you've seen what you have to do to scale some numbers, of course you want ADAM to do the work. The outline of a module for scaling is the following:

1. Check all the data to find out the biggest entry (call that number *be*).

2. The scale factor is the size of the graph divided by the biggest entry. (This is *sz/be* if *sz* is the size.)

3. Have ADAM multiply all of the entries by the scaling factor.

The program can't decide the size of the graph because it is determined by how you will use the scaled data or on what device (screen or printer) the graph will be displayed.

ADAM must use all the data to find the largest entry. This takes one pass through the data. After it finds the largest entry and has been told the size of the display, it can decide the scale factor. We then have to ask

ADAM to make a second pass to do the scaling and the plotting. (If the program used dynamic dimensioning, you might also count the number of data items in the first pass.)

Here's a typical scaling module:

```
4497 REM ************************************************
4498 REM   scaling module for data >= 0
4499 REM ************************************************
4500 max = -1E38: REM initialize the maximum value absurdly
small
4510 FOR i = 0 TO num: REM num is the number of entries
4520 IF d(i) > max THEN max = d(i)
4530 NEXT i
4540 sc = sz/max
4550 FOR i = 0 TO num
4560 d(i) = sc*d(i): REM scale everything
4570 NEXT i
4580 STOP
4590 REM ************END SCALING*****************************
```

Nothing in this module is hard. Notice that we had ADAM compute the scaling factor *sc* outside any loop. This is just common sense (and good programming). ADAM is fast, but the scaling factor is one number—it should be computed just once. If it were computed inside a loop, then it would be recomputed in each pass through the loop. It would be a waste of time to have ADAM compute it again and again. In this module the number of entries, *num*, must come from an earlier part of the program.

The scaling we've just shown you uses zero as the reference value. A *reference value* is the number on which all other information is based. For example, in our bar graphs the bars are very small when they represent numbers near zero. A bar representing zero units is zero units high. There is nothing sacred about having bars of height zero represent zero units. They could represent five units, fifty, or fifty million. This is a different kind of scaling called relative scaling. *Relative scaling* means that the reference value is not zero. (This is similar to the relative coordinates in the last section.) By using a different reference value, this method can be used to show only the differences between the data.

Suppose we wanted to plot a bar graph in a 25-unit space, and the numbers to be represented were 1000, 995 and 990. Using the scaling module, each bar would end up being 25 units high because the scaled numbers (after round-off) are 25, 25, and 25. It would look very different if we didn't use zero as the reference. If the reference height is 975, then the first bar

would be 25 units high, the second 20, and the third 15. Changing the reference value to 975 exaggerates the differences between the values. Whenever you look at a bar graph (or most any type of graph), check the reference values to see whether the differences have been exaggerated by this trick. This trick is used too often to prove a point—even if people have to be deceived to do it.

Finally, suppose you wanted to figure out the proper scale for a typewriter graph. The paper is usually 8½ inches by 11 inches with 65 rows and 80 columns. Each row, then, is 11/65 inches long. Each column is 8.5/80 inches wide. These two numbers are used as the size in the scaling module (see program on page 277). You have to scale in the horizontal and vertical directions separately.

Self-Check 4

A. Questions

1. What is the purpose of scaling?

2. Suppose a map is scaled so that 1 inch corresponds to 2 miles.

   a. How long would a 1/2-mile road look on the map?
   b. How long would a 2000-mile highway be?
   c. How long would a 100-foot house be?

   The remaining questions are about the scaling module (see program on page 273).

3. In line 4500, *max* is set equal to -1E38. Why? What might go wrong if we set *max* = 50?

4. How would you modify the program to handle data that is no bigger than 0?

5. How would you modify the program to use a reference value other than zero? (This is important if the data can be positive or negative.)

B. Programs

1. Incorporate a scaling module into the bar-graph program (see program on page 168).

2. Rewrite the typewriter graphing programs (on page 273 and 277) to use scaling.

Summary

In this chapter we introduced you to arrays, which are like lists but have more subscripts. We showed you another variable type, the integer variable, and looked at all the types of variables and how much room they take up in ADAM's memory. We discussed scaling and relative coordinates. Programs to print graphs and paint low-res graphics were written. The only command introduced in this chapter is:

FRE(0) shows you the amount of space in memory which is not currently assigned (is free).

BUZZwords

ALIGN

ARRAYS

FLOATING POINT VARIABLE

INDEX

IDENTIFIER

INTEGER VARIABLE

NUMERIC ARRAY

REAL VARIABLE

REFERENCE POINT

REFERENCE VALUE

RELATIVE COORDINATES

RELATIVE SCALING

REPRESENT	STRING ARRAY
SCALE FACTOR	STRING VARIABLE



Subroutines

In this chapter you will see a new way to organize your programs. We will show you four commands:

GOSUB	RETURN
ON—GOSUB	POP

which, when combined with modular programming, make your programs more flexible and powerful. These new commands can be thought of as GOTOs with a memory — after they send out for processing, they return for more orders.

Subroutines and Top-Down Programming

Programs work best when they are organized like the households of the very rich. The very rich don't often do housekeeping, run errands, or take

care of similar chores. Instead, they have various subordinates who take care of these details for them.

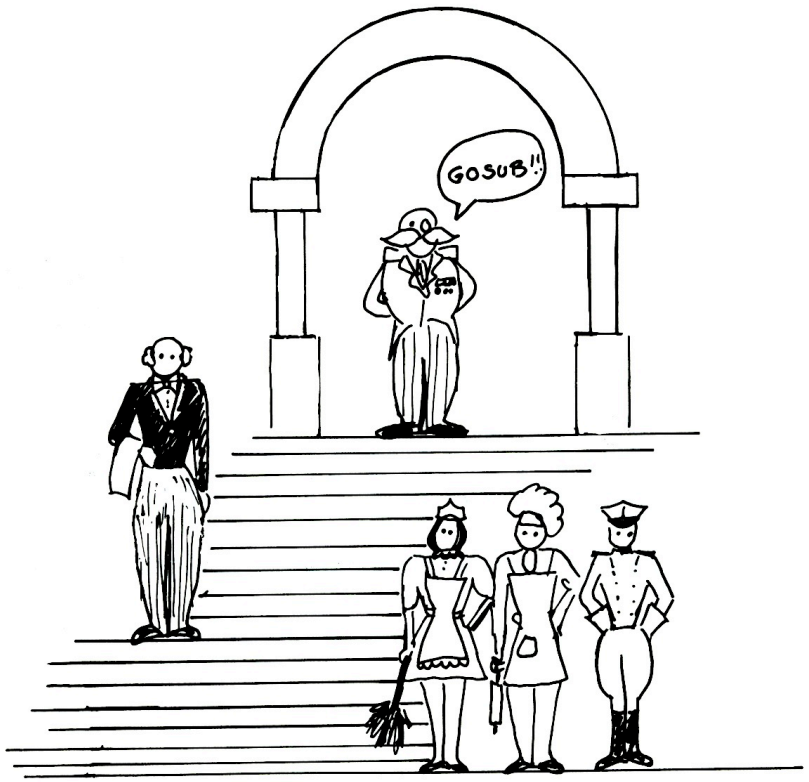


Fig. 12.1

Computers like the same kind of organization. A powerful and flexible program allows the *main* or *controlling* module to have servants. When something needs to be done, the main module should send out for it. After the operation is completed the main module resumes control. The commands in SmartBASIC that allow you to do this are:

```
GOSUM                (to send control out)
RETURN               (to return control)
```

GOSUB is the subroutine call command. (*Subroutine call* is the buzzword for using the command GOSUB.) GOSUB works almost like GOTO. GOSUB *line number* sends control to the module starting at that line. In writing programs, the biggest difference between using a GOTO or using a GOSUB is how the modules end. All our previous modules ended either with another GOTO statement or with an END command. When control is transferred to a module using the command GOSUB, the last statement in the module must be the command RETURN. RETURN transfers control to the statement following the (GOSUB) statement that sent ADAM to the module in the first place. So GOSUBs are like GOTOs but they have a memory. They remember where they came from and let ADAM return to a place in the program that *doesn't* have to be specified beforehand. For example, suppose you wrote a subroutine, between lines 1000 and 1500, that paints a picture on the screen. The actual program might look like:

```
 50 GOSUB 1000
      .
      .
      .
500 GOSUB 1000
      .
      .
      .
1000 ┐
  .  │   paint a beautiful picture on the screen
  .  │
1490 ┘
1500 RETURN
```

You can use something like the program fragment above to paint a picture on the screen. Line 50 tells ADAM to paint a picture. After the first picture is painted ADAM processes the next statement after line 50. After that, ADAM continues processing more statements until it reaches line 500. This line again calls the subroutine that paints the picture. Without

the GOSUB command, it would have been very hard to use the same module (lines 1000–1500) to paint the picture.

If it still doesn't seem like subroutines are an improvement over GOTOs, here are three important advantages they have:

1. A module ending with a GOTO is hard to use more than once in a program. A subroutine can be "called" from any point in a program. After the subroutine completes its task ADAM immediately "returns" to process the statement following the GOSUB. This means subroutines can easily be used many times.

2. Subroutines are computer friendly. On most computers a subroutine call works faster than a GOTO.

3. Subroutines are completely separate parts of a program. You can find subroutines that accomplish different tasks in reference books, on disk, or on tape. After you learn to use subroutines, these collections can be used as a reference library. Look up the subroutine you need and copy it into your program.

   (The SmartWriter word processor makes it easy to piece together parts of a program. First save the needed subroutines as separate files. Make sure that the line numbers for the subroutines don't overlap. Next put ADAM into its word processor. Now that ADAM is a word processor, you can use it to join together the subroutines saved in different files. For more details see the section of the word processor manual supplied with ADAM.)

An elegant and efficient program needs a GOTO with a memory—and that's the GOSUB.

To see how these commands simplify writing a program, look at the following two ways to print a large *X*. The one on the left is the program from chapter 8 (see page 185).

```
10 REM X without subroutines
20 PR#1
30 FOR i = 0 TO 23
40 PRINT SPC(10+i);"XXX";
SPC(46-2*i);"XXX"
50 NEXT i
60 REM now the bottom part
70 FOR j = 23 TO 0 STEP -1
80 PRINT SPC(10+j);"XXX";
SPC(46-2*j);"XXX"
90 NEXT j
100 PR#0
110 END
```

```
10 REM with GOSUB
20 PR#1
30 st = 0:tst = 23:tep = 1:GOSUB 100
40 st= 23:tst = 0 :tep = -1: GOSUB 100
50 PR#0
60 END
100 FOR i = st TO tst STEP tep
110 PRINT SPC(10+i);"XXX";SPC(46-2*i);
"XXX"
120 NEXT i
130 RETURN
```

Both programs print a large *X*. The program on the left used two essentially identical FOR-NEXT loops. Having two separate program fragments that do the same thing is a waste of effort (and makes the program harder to follow).

The program with GOSUBs is clearer and more concise. It has a four-line control module. Lines 30 and 40 are the only lines in the control module that do anything. Each line sets the starting, test, and step value for the FOR-NEXT loop and sends that information to the subroutine (lines 100 to 130). The subroutine uses this information as raw material for its job. The first time we call the subroutine the starting value *st* is 0, the test value is 23, and the step is -1. On the second call we reverse the order.

The first time we ask ADAM to GOSUB (go to the subroutine), it prints the top half of the *X*. Then it reaches line 130, which contains the RETURN command. RETURN transfers control to the statement following the GOSUB. This is line 40. So after the RETURN, ADAM processes line 40. It resets the stage (changes the values of *st*, *tst*, and *tep*) and then GOSUBs again to print the bottom half. The most important thing to notice is how lines 30 and 40 set the stage for the subroutine by telling the messenger variables how to set up the FOR-NEXT loop. This "stage setting" is common when using GOSUBs.

If you try to write the program on the right using a control module and a single printing module with one FOR-NEXT loop, you will run into a big problem. The last statement in the printing module has to be a GOTO command. But which line in the control module should ADAM go back to? There are two different places that control must go back to. The first has to print the lower half of the *X* and the second must end the program. This illustrates the first advantage of subroutines—they can be used more than once.

> WHEN YOU TELL ADAM TO GOSUB (TO A LINE NUMBER), IT REMEMBERS THE PLACE IT CAME FROM. AFTER THE COMMAND RETURN, ADAM GOES BACK TO THE NEXT STATEMENT FOLLOWING THE PREVIOUS GOSUB.

To prevent difficulties, the GOSUB should be the last executable statement on that line. (If you have rrom it's a good idea to have a REM statement following the GOSUB to indicate what the subroutine is supposed to do.)

END commands may have seemed unnecessary in the past. When

ADAM had no more commands to obey, a program automatically ended. The END command on line 60 is *very* important. Without this command ADAM would "fall into the subroutine." To see what this means, suppose you forgot to include line 60. Then ADAM would continue processing the lines in order until it got to line 140. Having gotten there, it would try to obey the command RETURN. However, ADAM doesn't know to where it should return. The result is the error message:

```
RETURN without GOSUB Error in 140
```

and your program bombs. When you use subroutines, the program must have an END or some other statement that will prevent ADAM from falling into the subroutines. Usually the control module is the best place for the END statement.

Another place subroutines can clean up your programs is in IF-THEN statements. As we mentioned in chapter 5, you often want to do more than one operation if a single condition is true. We know this can be done by collecting the different commands on the same line — each one separated by a colon. However suppose you wanted to do ten things when a single condition was true. Using colons would lead to incredibly complicated lines like:

```
IF_____ THEN COMMAND 1:COMMAND 2:COMMAND 3:
COMMAND 4 ..........:COMMAND 10
```

First of all, even *if* ADAM could accept this monstrous line, no one else will ever be able to understand your program. And in a day or two, you won't either. Isn't it easier to enter this?

```
IF ____ THEN GOSUB (someplace in the program
                    that does 10 things)
```

After ADAM does your ten commands it will return for more.

Subroutines are also useful for printing patterns. Suppose we want to print the word "HELLO" using *s. The module that does this might look like:

```
1010 PRINT "*   *  ****  *     *      *"
1020 PRINT "*   *  *     *     *    *   *"
1030 PRINT "***** ****  *     *   *     *"
1040 PRINT "*   *  *     *     *    *   *"
1050 PRINT "*   *  ****  ****  ****   *"
```

Since this module takes 5 lines and 30 columns, we could print it four times on the screen. This is easy. Make it into a subroutine by adding:

```
995 REM
996 REM
997 REM ******************************************************
998 REM  subroutine to print "HELLO"
999 REM ******************************************************
1000 VTAB q:HTAB 1
          .
          .
          .
1060 RETURN
```

The control module is a simple FOR-NEXT loop:

```
10 REM a HELLO program
20 HOME
30 FOR f = 1 TO 19 STEP 6
40 q = f: GOSUB 1000
50 NEXT f
60 END
```

We called this a HELLO program as a way of introducing another of ADAM's nice features. Any program that is saved under the name HELLO on your SmartBASIC tape will automatically be run after you load the SmartBASIC tape. A program like this (often called a *turn-key program*) is the one exception to the rule against saving programs on your SmartBASIC tape. You can write any HELLO program you want—just make sure that it is saved on the SmartBASIC tape with the name *HELLO*. The capital letters are necessary (see chapter 16).

We could have used the *f* in both the FOR-NEXT loop and the sub-

routine. We didn't because it is a *very bad idea* to use the same variable in a subroutine as in the controlling module. If you can't initialize the variable at the beginning of the subroutine, then you must be very careful. BASIC doesn't have a way of distinguishing between variables that are used in the main part of a program and those used in subroutines. Look at the following program:

```
10 REM a terrible use of a variable
20 FOR s = 10000 TO 50000 STEP 500
30 GOSUB 1000
40 NEXT s
50 END
995 REM
996 REM
997 REM ********************************************************
998 REM  subroutine to compute taxes
999 REM ********************************************************
1000 LET s = s -.067*s
1010 PRINT "Your net salary is ";s
1020 RETURN
```

ADAM never gets to the end of the FOR-NEXT loop because the subroutine changes the value of *s* in each pass through the loop. (The counter variable is changed in the body of the loop.) This example may seem contrived, but the error we are trying to illustrate is common. Be careful that your subroutines *pass* the exact information you want back to the rest of the program - and that nothing else slips through by mistake. We avoid these problems by making a table that carefully outlines which variables go in which modules. That table stays by our side as we change the outline for our program into SmartBASIC code. In computerese, variables that contain information passed to and from a subroutine are called *global variables.* Those that are used only inside subroutines are called *local variables.* In the program on page 284, the messenger variables *st, tst,* and *tep* are the global variables, while the *i* used as a counter is a local variable.

It is precisely because BASIC has no built-in way of distinguishing between global and local variables that professional programmers dislike BASIC. Other programming languages such as FORTRAN and PASCAL make you say which variables are global and which are local. In BASIC you have to do this bookkeeping yourself.

Self-Check 1

A. Questions

1. What's wrong with the following program?

```
10 INPUT a
20 newa=a: GOSUB 1000
30 PRINT newa
1000 IF newa < > 0 THEN newa = -newa
1010 RETURN
```

2. What are the global and local variables in our HELLO program?

B. Programs

Try to simplify programs from earlier chapters using GOSUB's. For instance:

1. The program to draw an abstract design (bottom of page 209).

2. The bar graph program (see program on page 168).

More on Subroutines

A good way to think about subroutines is that they let you make up your own commands in SmartBASIC. There is no command to print a row of thirty-one *'s, so we use a subroutine instead. If you can't find a command in a SmartBASIC reference manual to do what you want, write a subroutine to do it.

SmartBASIC has graphics commands to draw horizontal and vertical lines but none to draw frames. If we write a subroutine to draw a frame, we can incorporate that subroutine into any program that needs it. In order to have ADAM draw a frame, ADAM needs a color, the size of the sides and the location of the frame. Here is the module:

```
991  REM ************************************************
992  REM frame drawing subroutine
993  REM ************************************************
994  REM global variables used - co,row,kol,leng,wid
995  REM local variables used - none
996  REM ************************************************
997  REM co is color, row is row, kol is column, wid is
998  REM width, leng is length
999  REM ************************************************
1000 COLOR = co
1010 HLIN kol,kol+wid AT row
1020 VLIN row,row+leng AT kol
1030 HLIN kol,kol+wid AT row+leng
1040 VLIN row,row+leng AT kol+wid
1050 RETURN
```

You may have noticed that we haven't had ADAM check whether data passed to the subroutine is allowable. We just assume, for example, that the value of *kol* is between 0 and 39. This is not carelessness on our part. Ideally a subroutine should be designed to accomplish exactly one job. We are assuming that before the values of *kol* (or anything else) are passed to the subroutine they are checked (possibly even by another subroutine!). We left the command GR out of the subroutine because this command erases everything on the screen—which is certainly not something this subroutine was designed to do.

This subroutine can print a frame at any time and in any size. Just include a line like:

```
kol = ?:  row = ?:  co = ?:  leng =?
GOSUB 1000
```

and replace the question marks with numbers (or numeric expressions) that determine the color, size, shape, and location of the frame.

The various REM statements at the beginning of the subroutine contain information about what the subroutine does and what information it needs to operate. Explaining how a program works, the variables needed, etc., are the keys to good documentation. (*Documentation* is the buzzword for what accompanies, or is contained in, a program to explain it.) The listing of local and global variables as REM statements gives the programmer the information necessary to use the subroutine—in any program. (It also lets you check that the names of variables in the main program don't conflict with the ones in the subroutine. If they do, you can make whatever changes are necessary.)

Here's another example: lots of our programs have *error traps.* These are usually tests to prevent a program from asking ADAM to execute an illegal command. Examples are to test that a color code is between 0 and 15 or that the program doesn't try to force ADAM to plot a block outside the graphics area. Since sounds are an excellent way to get people's attention, it is a good idea to ring a bell when an error is detected. Here's a bell-ringing subroutine:

```
9994 REM ******************************************************
9995 REM  bell ringing subroutine
9996 REM ******************************************************
9997 REM global variables used - none
9998 REM local variables used - b9
9999 REM ******************************************************
10000 FOR b9 = 1 to 10
10010 PRINT chr$(7)
10020 NEXT
10030 RETURN
```

This subroutine rings the bell 10 times by printing *CHR$(7)* each time it is called. There is one new idea introduced in this subroutine. We used a strange name for our variable to minimize the possibility that any other variable (in a program that calls this subroutine) have the same name and have its value messed up by the subroutine. You can change this subroutine so the number of times it rings the bell is determined by a global variable passed to it. To do this, introduce the variable *chime* and change line 10000 to:

```
10000 FOR br = 1 TO chime
```

The RETURN on line 10030 sends processing back to the statement following the GOSUB. Here's an example in pseudo-code:

```
color code test
IF co >= 0 or c <= 15 THEN o.k. test
GOSUM bell reminder
INVERSE:PRINT "color code not allowed"
```

You might use a subroutine in lines like:

```
PRINT "Do you want to continue"
INPUT "Enter yes or no ";yn$
IF yn$ = "yes" THEN GOSUB (someplace)
GOTO  (probably back to controlling module or
      the end)
```

These lines are written in pseudo-code, not in SmartBASIC. This shouldn't cause problems because the whole point of pseudo-code is to have something that is easily translated into SmartBASIC statements.

There is one important variant of GOSUB called the ON—GOSUB command. We spent a lot of time showing you the ON—GOTO command in chapter 5. ON—GOSUB works exactly the same way except ADAM doesn't GOTO, it GOSUBs. Menu-driven programs work better with ON—GOSUB. Here is an example in pseudo-code:

```
PRINT 1. . . . First MENU item
      2.       Second item
      3. . . . .
               .
               .
```

(continued)

```
INPUT "Select Menu Number please"; x
ON 1 GOSUB (somewhere)
On 2 GOSUB (somewhere else)
```

and so on.

Using subroutines correctly is the last major hurdle left in mastering BASIC. Developing the habit of thinking of each separate operation as a subroutine encourages what is called *top-down programming.* Top-down programming is an extension of modular programming. Like many programming techniques, it is just common sense. The idea behind modular programming is that a program should be developed from the general to the particular. Start with one big job and then, in stages, break it down into smaller and smaller pieces. This lets you see the forest even when there are lots of trees. The first outline lists the jobs that have to be done. At each stage, the problem becomes easier to solve. By the final step, each part should be easy to program. That's when you start the actual coding. The rule of thumb for writing really long programs (those with hundreds of lines) is that you keep on refining until the parts to be written are no longer than a page (about twenty lines). Some of the reasons for using this method are:

1. Individual tasks are kept separate so they can be coded by different programmers (if necessary).

2. Often the individual tasks can be accomplished by standard subroutines.

3. And most important: you never have to deal with too many details at the same time.

Some people keep something like figure 12.2 (page 294) by their sides. The placement of these boxes indicates the relationship between the various parts of the program and the final level indicates operations close to what a computer can be asked to do. When you're programming in SmartBASIC, the final level usually consists of subroutines.

In top-down programming, requests for actions pass down the levels and only results of operations pass up. This is the reason it is an extension of modular programming. In top-down programming, not only should you

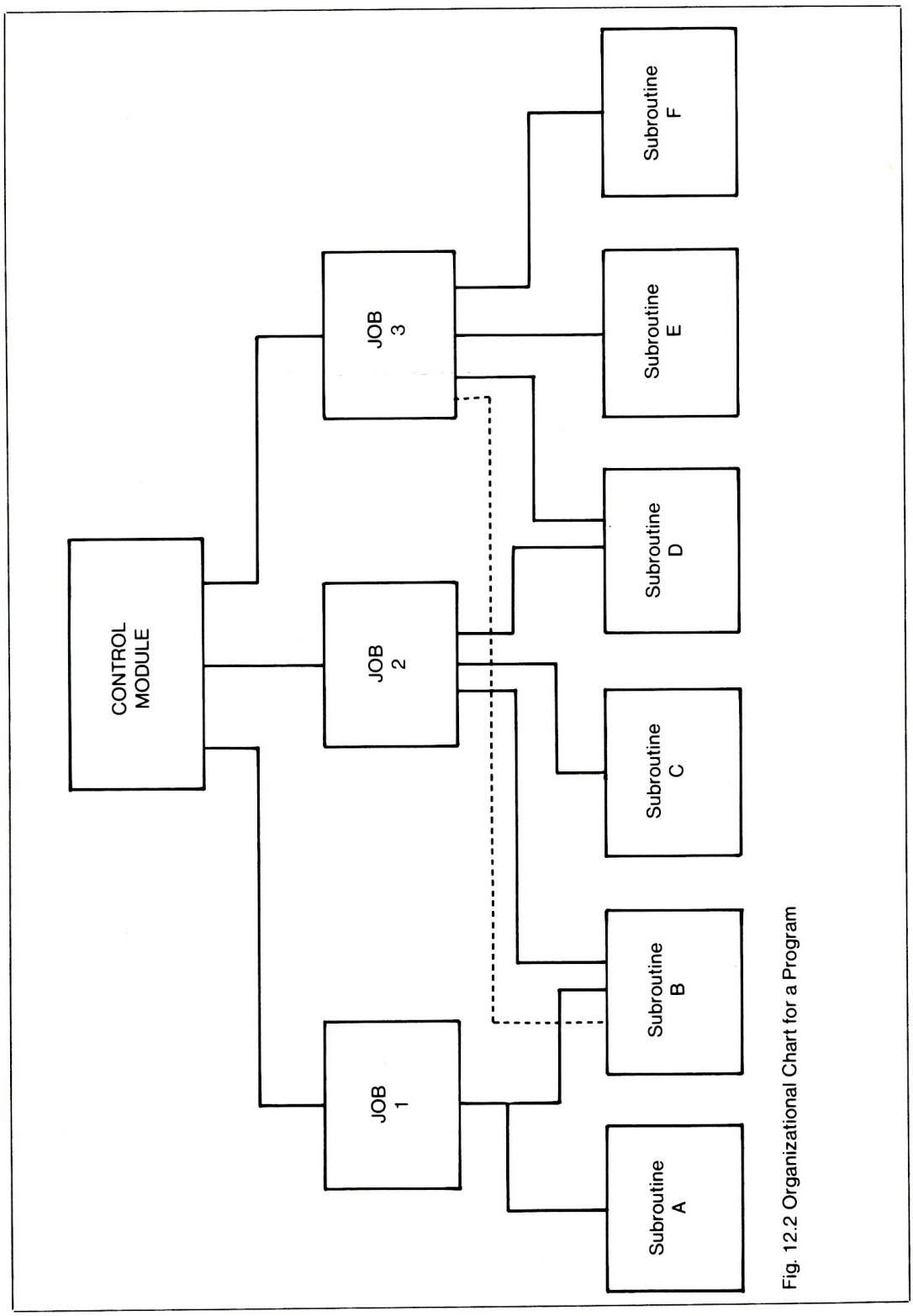


Fig. 12.2 Organizational Chart for a Program

write the program in modules, the modules should be developed and written starting at the top level, then at the next level, and so on. The final ingredient in good top-down programming is the avoidance of GOTOs. GOTOs can be used to cut across levels and this is a bad idea. (It can even occur accidentally and then is very difficult to find and correct.) Professional programmers have found that the fewer GOTOs in a program, the more likely it is to run correctly and (if necessary) the easier it is to debug or change.

Most of a professional programmer's time is spent modifying programs written by other people. Imagine a big program that wasn't written originally in a top-down manner. You are hired to make a small change at one place in the program and you change (or possibly introduce) a few variables. Because the original program wasn't written "top-down," the little change you make *could* foul up the whole program! This kind of disaster was very common until the late 1960s or early 1970s. Companies first spent millions of dollars having programs modified; then they spent more money trying to anticipate all the problems that the changes might cause; and then they hoped that more time and money would fix the possible problems that they anticipated. After all of this, they still could never be certain that the programs were free of bugs. Top-down design stopped these horrors completely; programs still have bugs, but these bugs don't cause epidemics. If you fix a small module in a giant top-down program, you know how it will affect everything on a higher level. So it can't unexpectedly damage anything on a higher level.

Since a GOTO is an invitation to jump layers, some people have advocated banning them completely. This, however, is going too far. A GOTO can certainly be used downward in a program and, in BASIC, this is absolutely essential. Just use them sparingly; too many GOTOs leads to "spaghetti" programs (see Self-Check 1 in chapter 5) and difficulties. Because GOTOs were over-used in the early days of FORTRAN and BASIC programming, many programmers think of them as four-letter words.

Self-Check 2

A. Questions

1. How is top-down programming different from modular programming?

2. Which variables in the programs on page 284 and the bottom of page 287 are global and which are local?

B. Programs

1. Write a subroutine to draw the letter *L* in a size set by a global variable.
2. Write a program to print the following pattern all over the screen.

```
$
$$$
$$$$$
```

Subroutines and User-Defined Functions

In chapter 9, we showed you how to define your own functions with the command *DEF FN.* These functions were useful but they won't accomplish everything you might want to do. For example, the DEF FN command only allows you one line to make the definition, and sometimes you need more room. Another problem is that the function can depend on only one value—the functions are one-variable functions. Subroutines let you get around all these problems. With a subroutine, you can define much more complicated functions with any number of variables.

Suppose you wanted to define a function that took any two numbers you gave it and returned (*returned* is the buzzword for give back) the largest. As a subroutine this is easy; as a function, it's impossible.

```
991 REM ************************************************
992 REM  subroutine to find the larger of two numbers
993 REM ************************************************
994 REM global variables used-t1,t2,larg  *
```

(continued)

```
995 REM local variables used-none          *
996 REM ************************************
997 REM t1,t2 are the numbers              *
998 REM larg passes back the larger        *
999 REM ************************************
1000 IF t1<t2 then larg = t2: GOTO 1020
1010 larg = t1
1020 RETURN
```

Of course we've written much more complicated modules that do even more. The one that found the maximum element of a list is an example. What we wanted to stress here is that the variables *t1* and *t2* can be used much like the dummy variable in the DEF FN command. This time you use it in lines that might look like:

```
2050 t1 = x:  t2 = y:  GOSUB 1000
2060 PRINT larg ;" is the larger"
```

It's not hard to use a subroutine as a function in this way. Just be careful to initialize the information sent to the subroutine. You might have noticed that we used a slightly different format for this subroutine. We haven't changed the commands, but only the REM statements. This only changes the way the subroutine is printed (listed), not how it runs. There is nothing sacred about the format of a program. It must present the code in a readable and useful manner. Different subroutines are separated by blank lines (that is, only containing REM commands) to make them easier to locate.

Using subroutines sometimes simplifies the translation from outline to program. Here is another way to outline (in pseudo-code) the "sort" program from chapter 10.

1. Get the list.
2. GOSUB to find smallest on the list.
3. Put the smallest in the first position.
4. Shrink the list by using all entries but the ones already ordered.
5. If the list is finished, then print the rearranged list.
6. Otherwise go back to step 2.

This outline replaces the nested FOR-NEXT loop originally used by repeated use of a GOSUB. If you think of finding the smallest as the simplest operation, then this translation is more direct than the one in chapter 10. Never forget that there is almost always more than one way to program a solution. Use whatever method makes you most comfortable (but also talk to other people about your methods. You may find that someone has found a really neat trick for doing something and that you can use a similar trick in your programs.)

Subroutines can also be used in a general purpose program for preparing tables of values of mathematical functions. (If you don't need such tables, you might want to skip to the next section).

Here are some examples of mathematical functions:

$y = 3x^2 + 2$ (in computerese, y = 3*X*X + 2)

$y = x^4 + x^3 + 1$ (in computerese, y = x ^ 4 + x ^ 3 + 1)

$$y = \frac{x + 1}{x^2 + 2}$$ [in computerese, y = (x + 1) / (x ^ 2+2)]

(Examples 1 and 2 are polynomials and example 3 is a rational function.)

When you write down a polynomial, you have to add powers of x multiplied by numbers. The highest power of x that appears is called the *degree* of the polynomial. The numbers that multiply powers of x are called *coefficients.* In the first of the examples shown above, the degree of the polynomial is 2 because 2 is the highest power of x, that appears. The coefficient of x^2 is the number multiplying x^2, which is 3. The coefficient of x^1 is 0. The constant term is 2, which is the coefficient of x^0. (Remember that anything raised to the power zero is 1.)

It can be useful to have a program that prepares tables of the values of the variable x and the corresponding values of y for polynomial and rational functions. Enter the coefficients, give the range for the table, and ADAM does the rest. With subroutines this program isn't hard, without them it is quite complicated.

A module to enter the coefficients of polynomials is easy:

```
997 REM ************************************************
998 REM  subroutine to enter the coefficients
```

(continued)

```
999 REM ************************************************
1000 INPUT "What is the degree of your polynomial? ";deg
1010 DIM coef(deg)
1020 PRINT "Remember to enter 0, if the coefficient is zero"
1030 FOR q = deg TO 0 STEP -1
1040 PRINT "Enter the coefficient of x^";q;" ";
1050 INPUT coef(q)
1060 NEXT q
1070 RETURN
```

Notice that the coefficients of the different powers of x are stored as a list.

The module to compute the value of the polynomial is a bit trickier. We start out with a variable called *xval* taken from outside—this is the value of the variable that we will use in the computation. Using *xval*, we can compute the value *yval* of the polynomial by adding *xval* raised to different powers multiplied by the corresponding coefficients. This method for evaluating a polynomial mimics the way we write it as a sum of simple terms. As an example, for the polynomial:

$y = 3x^2 + 2$	(in computerese, y = 3*X*X + 2)
	Start with yval = 0
coef(2) = 3	First step: yval = yval + coef(2)*xval ^ 2
	(yval = 3*xval ^ 2)
coef(1) = 0	Second step: yval = yval + coef(1)*xval ^ 1
	(yval = 3*xval ^ 2)
coef(0) = 2	Third step: yval = yval + coef(0)*xval ^ 0
	(yval = 3*xval ^ 2 + 2)

As we've said before, the best way to figure out how to write a program is to work through enough examples by hand. Here's the module:

```
1990 REM ************************************************
1991 REM  polynomial evaluation subroutine
1992 REM ************************************************
1993 REM global variables-deg,xval,yval,coef(q) *
1994 REM local variables q                      *
1995 REM ***************************************
1996 REM the coef(q),deg and          *
```

(continued)

```
1997 REM xval come from outside       *
1998 REM the yval is returned         *
1999 REM ******************************
2000 yval = 0: REM intialize the variable yval
2010 FOR q = deg TO 0 STEP -1
2020 yval = yval + coef(q)*xval^q
2030 NEXT q
2040 RETURN
```

Tricks of the Trade We've mentioned in passing that subroutines can and usually will call other subroutines. Even more interesting is that they can call themselves. When a subroutine calls itself, people say that the program uses *recursion.* Recursion isn't used much in BASIC, but it is used frequently in other popular languages such as LOGO, LISP, or PASCAL. It is interesting to see how this can be done (and, of course, why you would want to do it!). Suppose ADAM encounters the line:

```
10 GOSUB 1000
```

It stores the number *10* in a list somewhere in its memory. Now suppose the subroutine on line 1000 contains the line:

```
1050 GOSUB 2000
```

then the list grows and the number *1050* piles on top of *10*. If you continue this process ADAM will build up a stack. (*Stack* is the buzzword for a list of places or line numbers in the program to which control must return. We're simplifying this idea a bit by treating each line as a separate statement.) Eventually ADAM encounters the command RETURN. It looks at the stack in its memory and then RETURNs to the place following the top entry in the stack. That number is thrown away. The next RETURN uses the next stack entry, and so on. So if figure 12.3 represents ADAM's stack at some time, the first RETURN sends control back to the statement after the GOSUB on line 5000. The next RETURN ADAM encounters sends control to the statement after the GOSUB on line 4000. If there are enough RETURNS in the programs, control eventually passes back to the statement following line 10.

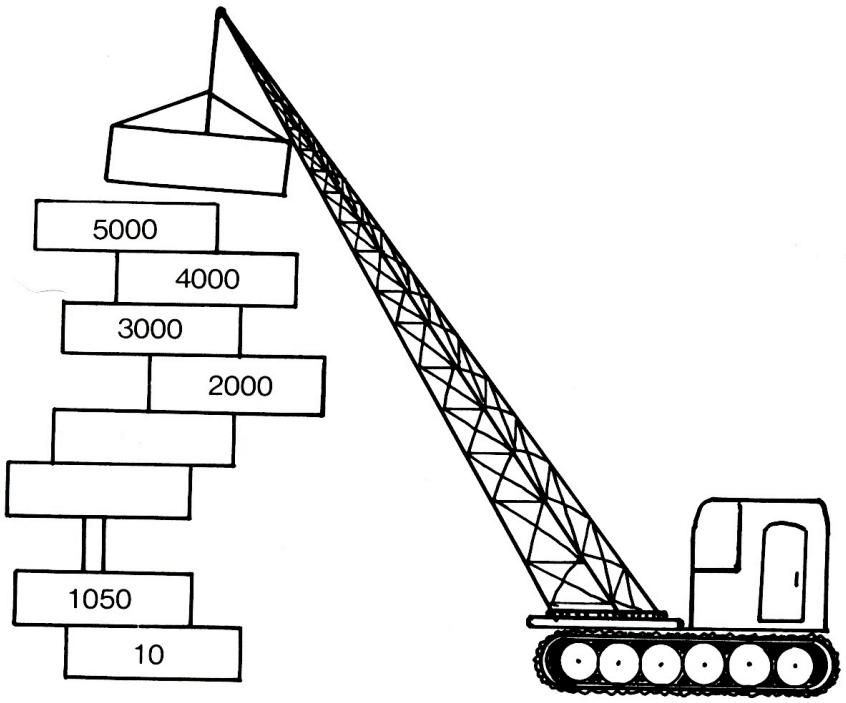


Fig. 12.3

This explains why ADAM gives an error message when you "fall into a subroutine." It sees the command RETURN but the stack is empty—so it doesn't know what to do. When ADAM gets frustrated, it sends you error messages.

The stack doesn't care what line numbers it contains. This makes recursions like the following possible:

```
10 REM a recursion
20 a = 10
30 n= a: GOSUB 1000
40 END
1000  PRINT n
1010 IF n <= 0 THEN 40
1020 n = n - 1
1030 GOSUB 1000
1040 RETURN
```

Run this program. It is a silly way to print the numbers from 10 to 0. Silly as it is, it does illustrate the idea of a stack and how recursion works. ADAM builds up a stack 10 levels high before it exists via line 1010. If you use recursion you *must* have a way of exiting. ADAM can only hold 31 line numbers in its stack before it complains, prints:

```
Stack Error
```

and your program bombs. You can see for yourself how deep ADAM's stack is by trying different values of *a* in line 20. This program doesn't show why recursion is a feature of more sophisticated programming languages. We have already mentioned that subroutines can be thought of as operations or functions that you can add to SmartBASIC. Certain functions are most naturally defined in terms of *themselves.* One good example is the *factorial function* from chapter 6. Recall that this function gives the product of the numbers from 1 to N. Defined in terms of itself, it is:

```
FACT(1) = 1
FACT(N) = N * FACT(N-1)
```

Since a computer can calculate functions by building up a stack, many languages (for example, LISP) are incredibly powerful because they are set up to perform this type of recursion.

When you use many subroutine calls (as we said before, this buzzword means what ADAM does after a GOSUB), sometimes you want ADAM to forget where it is supposed to return to and go to a different place. A GOTO can't work because this won't change the stack. If you use a GOTO, any other RETURN command encountered later in the program automatically sends ADAM to the place you didn't want to go. If you use the command POP, ADAM forgets the top line number in the stack. ADAM "pops" the number off the top of the stack. The effect of the POP command is that ADAM treats the first RETURN command as though it were the second, the second is treated as though it were the third, and so on.

The POP command is important in recursions. For example, the program on page 301 has one bad feature. When the count gets up to 10 and control

passes back to line 40, none of the line numbers on the stack have been used. They're all sitting in the stack, taking up space, and could eventually cause problems. We should remove them from the stack. However, RETURN and POP are the only ways to get line numbers off the stack (without ending the program). In this recursion, no RETURN statement is ever processed. The easiest way to remove the line numbers in the stack is to POP them as soon as they get onto the stack. You only have to change line 1000 to read:

```
1000 POP:  PRINT n
```

Self-Check 3

A. Questions

1. Why are subroutines more versatile then user-defined functions?
2. Why do we start with w = 0 in the evaluation module?
3. What happens when you POP all the line numbers off the stack and then your program needs to execute a RETURN? (Insert a POP command into one of the subroutines and see what happens.)
4. What happens when you try to POP too many line numbers off the stack? (Insert a few POP commands into one of the subroutines and see what happens.)

B. Programs

1. How would you write a module to evaluate a rational function?
2. Combine this module with the graphing program from the last chapter.
3. Write a module that will average the entries in a list.

Summary

In this chapter, we showed you a new way to organize a program. Using the command GOSUB, which acts like a GOTO with a memory, we showed you how to make your programs more powerful and flexible. The new commands were:

GOSUB	sends control to a specific line in the program.
RETURN	is the last statement in a subroutine. It returns control to the statement following the GOSUB that called the subroutine.
POP	changes the statement (or line number) that RETURN will send control to.
ON—GOSUB	is similar to the ON—GOTO, except that ADAM GOSUBs instead of GOTOs.

BUZZwords

DOCUMENTATION

GLOBAL VARIABLE

LOCAL VARIABLE

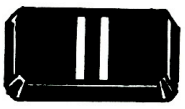
RECURSION

STACK

SUBROUTINE CALL

TOP-DOWN PROGRAMMING

TURN-KEY PROGRAM

INTERLUDE 

Testing Programs and Debugging

We've written lots of modules and programs to perform various tasks. How did we know that they would do what we wanted them to? Although we can't be absolutely sure, we did test each module or program enough to be reasonably confident.* This interlude explains some ways to write test modules. *Test module* is computerese for a special module written to test another module or program. Usually these modules are temporarily added to the program you are checking.

When the programs didn't work as expected, we had to find the errors and get rid of them. These errors are usually called *bugs* and the process of getting rid of them is called *debugging*. Ideally, a test module should not

*If you should discover any errors in the modules and programs presented in this book, please bring them to the attention of the authors c/o The Wiley Press.

only check for errors but also help you pinpoint the bugs. Unfortunately, writing good test modules is an art and not a science. What we can do is give you examples of test modules that we wrote and try to explain why we wrote them this way. After working through them, you'll be in a good position to develop your own style.

The first idea that comes to mind is to have the test module check all the possibilities. For example suppose we wanted to test the subroutine that finds the larger of two numbers as in the program on page 296. We might write a test module like:

```
12000 REM test module for the larger of two numbers
12010 INPUT "numbers to test?"; t1,t2
12020 GOSUB 1000
12030 PRINT "the larger of ";t1; " and ";t2; " is ";larg
12040 END
```

Then we enter RUN 12000 (recall ADAM lets you determine where to start running a program) and ADAM prompts us for two numbers. You should try the three different possibilities (first number bigger, first number smaller, and the two numbers equal) and if the correct number is printed each time, you can be reasonably confident that the subroutine works. This subroutine is so simple that it's easy to check all the possibilities.

Unfortunately, when a program is long and deals with many different situations or possibilities it's impossible to test each one; you have to be content with checking a reasonable sample. However the word reasonable is the key. There is a story worth keeping in mind. Supposedly a utility company had a long, carefully checked program to send out bills and follow-up bills, threatening notes, and finally to automatically cut off service if no response was received. Well, the story goes that one day someone went off to Florida for a vacation and shut off the electricity. The computer sent out a bill for $0.00—which obviously wasn't paid. After a while this poor person received a threatening note saying if he didn't pay $0.00 by the following Thursday his electricity would be shut off. A frantic call may or may not have succeeded in stopping the shut-off—the story doesn't say. If the story is true, then what probably had happened was the original programmer hadn't tested what the program would do if the bill was $0.00. This wasn't a "reasonable" possibility.

We want to show you the test modules we used for some of the more interesting programs from chapter 10.

Testing the "sort" module (program on the bottom of page 250) wasn't

hard using the random (RND) function. We first changed it into a subroutine by ending it with a RETURN statement. Then we made up a list.

```
3999 REM test module for the sort module
4000 DIM x(50)
4010  FOR a = 1 TO 50
4020  x(a) = RND(1): PRINT x(a)
4030  NEXT a
```

We had to GOSUB to the SORTing module with the command:

```
4040 GOSUB 2000
```

and finally to have the test module control the SORTing module we started the program with the command:

```
RUN 4000
4050 FOR a = 1 TO 50
4060 PRINT x(a)
4070 NEXT
```

When the list was printed in the right order we were confident that the sort module worked correctly.

Finally, suppose we wanted to test the "binary search" module (program on page 248). It is a little harder to develop a test module for this program. A good test module for this program needs:

1. An ordered list of reasonable size.

2. A number on the list to search for.

3. A number off the list so the search would fail.

What's more, you don't want the numbers to be too "sensible." If the numbers followed a particular pattern, the program might work for that

pattern and no others. To do all this we again used the random function. The list started out with a random number and then we had ADAM add more (positive) random numbers to it. Since the numbers are growing larger each time, they form an ordered test list. To choose a number on the list to search for, we randomly select one entry from the list. To find a number not on the list is a little trickier; one way is to average two consecutive entries on the list. Since the numbers are growing, this number can't be on the list.

```
REM test module for binary search
DIM test(1000)
LET u = RND(-1)
FOR i = 1 TO 1000
test(i) = test(i-1) + 10*RND(1)
NEXT
LET a= INT(1000*RND(1)) + 1
LET s= test(a): REM search for s - on the list
LET s1 = (test(a) + test(a-1))/2: REM s1 can't be on the
list
GOSUB 1000
```

We left out the line numbers because any test module is only temporarily added to the program. After the testing is completed this module is deleted. We set u = RND(−1) so that, if necessary, our sequence is repeatable. (Recall, from chapter 9, that using a negative seed gives a repeatable pattern.)

All is well if your testing succeeds—but sometimes it won't. Now you have to decide what is causing the problem(s). The results of a well designed test module will help. Suppose, for example the binary search module was able to find a number on the list but seemed to run forever if the number wasn't on the list. This pinpoints where the bug is. Look at the part of the module that checks if the search is complete.

Other mistakes can be more subtle—pinpointing the problem is that much harder. The test module gives you some idea of where the problem lies, but doesn't show you its exact position. You really have three tools available to find the errors in a program. They are:

1. Print out a listing of the program. Check the listing against your notes for the program. Do they agree? If not you probably made a typographical error. (It's quite hard to catch your own typos—ask someone else to help.)

If this wasn't the problem, you'll have to try following the program as each statement is processed. To do this, on a piece of paper analyze by hand what happens as each line is executed. Do only what each statement says. Get a good grip on what the program *should* do. After that keep the listing and your notes on the program's logic (in computerese, the *logic* of a program means the programming steps needed to make it work) by your side as you use the following two tools.

2. The easiest way to see how a program is running is to add lines that identify the values of variables at different lines, e.g.:

```
205 PRINT "I'm at line 205 the values of the
variable x is "; x
```

(To make effective use of these PRINT statements, you may want to reset the speed to slow down the display. By setting:

```
SPEED = 0
```

it's easy to see the changes on the screen as they happen. At the normal speed (255), the display may move too fast.)

Interspersing lines like this helps identify what ADAM is doing to the variables at different points in the program. Another way to do this is to halt the program by inserting the STOP command at various places (CONTROL/C doesn't work as well) and then to issue PRINT commands from direct mode. After you have identified the values of the key variables you can enter the command CONT and the program should resume. (Warning: Using commands other than PRINT will make it difficult to use CONT).

3. The last tool is the command TRACE. If you issue the command TRACE in direct mode, then enter RUN, ADAM will print—in order—the line numbers of the line it executes together with whatever output is supposed to be printed. A good way to see TRACE in action is to try a simple loop:

```
10 REM an example of TRACE
20 FOR i = 1 to 10
30 PRINT "I'm on the ";i;"'th loop"
40 NEXT
50 END
```

Issue the command TRACE in direct mode and RUN this program. The screen will look like:

```
#20 #30 I'm on the 1'th loop
#40 #30 I'm on the 2'th loop
#40 #30 I'm on the 3'th loop
#40 #30 I'm on the 4'th loop
#40 #30 I'm on the 5'th loop
#40 #30 I'm on the 6'th loop
#40 #30 I'm on the 7'th loop
#40 #30 I'm on the 8'th loop
#40 #30 I'm on the 9'th loop
#40 #30 I'm on the 10'th loop
#40 #30
```

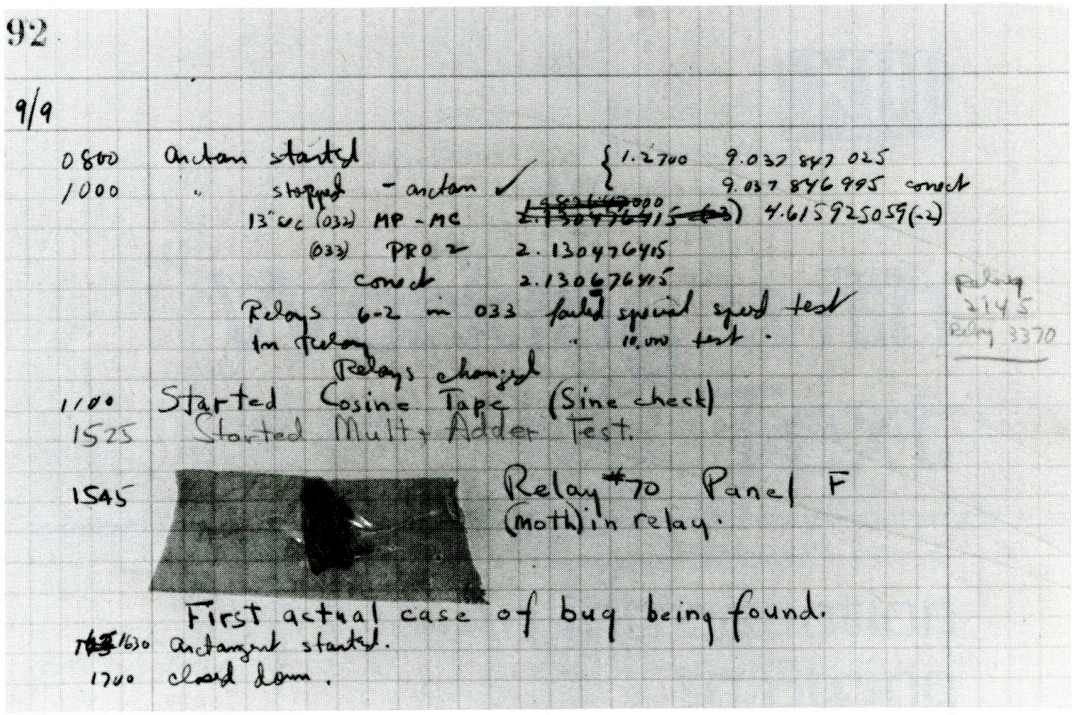
If you have understood the logic of your program, then tracing its operation is helpful. You try to find a place in the program where ADAM processes some line when it should be doing something else. Tracing a program is no cure-all. Before doing it be sure that you have outlined *by hand* the order in which the lines should be processed. Only then will tracing (combined with the earlier techniques) help you.

(You shouldn't get too cocky about how easy it can be to find the errors in a program. After all, ADAM is a complicated machine working at lightning-fast speed. The last two methods, inserting PRINT statements and using TRACE, can't show you everything that ADAM is doing. They help, but finding bugs in a program is hard work.)

The TRACE command stays on, so be sure to disable it with the NOTRACE command when you're finished with it.

We can't resist telling the story of where the term *bug* comes from. The MARK II computer (one of the first modern computers) used vacuum tubes, and tubes generate a lot of heat. For this reason the windows were often left open in the computer room. Tubes burn out frequently so a whole crew did nothing but test circuits and replace tubes. One day when the machine was down (*down* is the buzzword for not working; its opposite is the word *up*) they discovered that a moth had wandered in, been caught in a

relay and, in so doing, had caused a short circuit. The bug was removed, the relay repaired, and the MARK II was up again. Grace Hooper (she is one of the most important of the early computer scientists and was one of the most important forces in the development of high-level languages) remarked, "First successful debugging," and the moth was taped to the log book. It remains there today. Here's a picture of that page of the log book.



Summary

In this interlude we showed you methods for testing and diagnosing programs. The new commands were:

TRACE	causes ADAM to display the line number of a statement as it is processed.
NOTRACE	disables TRACE.

BUZZwords

DOWN

LOGIC (of a program)

TEST MODULE

UP



Working with Text

An Introduction to String Manipulation

In this chapter we will show you how to use SmartBASIC to manipulate words. These techniques will let you write words backward, make up "jumbles," center text on the screen, and more. You will learn the commands:

LEN	POS	VPOS	MID$
LEFT$	RIGHT$	VAL	STR$
GET	ASC		

and a new use for the +.

Putting Words Together

We've often created new numeric variables by adding together old variables. Frequently, we have to build strings by placing one string next to another. A computer looks at concatenation (*concatenation* is the buzzword for joining words or strings together) as a kind of addition. Once you know this, it shouldn't be surprising that the SmartBASIC command for joining two strings together uses the + sign. Suppose:

```
a$ = "Queen "
b$ = "Elizabeth "
c$ = "I"
```

Then *k*$ = *a*$ + *b*$ assigns the value "Queen Elizabeth" to the variable *k*$.

Letting *k1*$ = *a*$ + *b*$ + *c*$ gives *k1*$ the value "Queen Elizabeth I". You might notice that the assignment:

```
k1$ = a$ + b$ + c$
```

is the same as:

```
k1$ = k$ + c$
```

because *k*$ places the value of the variable *b*$ to the right of the value of *a*$; then the value of *c*$ is placed to the right of all that.

Finally, the assignments:

```
k2$ = a$ + b$ + c$ + c$
k2$ = k1$ + c$
```

give *k2$* the value "Queen Elizabeth II".

This works out nicely because we put spaces at the end of "Queen" and after the "Elizabeth" and left no space after "I". When ADAM sees a + it joins the two strings together—including any spaces—in exactly the order that they appear around the symbol +. So, unlike addition of numbers, order is important in concatenating strings.

You can't use + to join strings and numbers together. If you enter:

```
PRINT 3 + "-D"
```

you'll get an error message instead of the string "3-D". However, you can tell ADAM to:

```
PRINT "3" + "-D"
```

The quotes tell ADAM that *3* is to be regarded as a character, not as a number.

You can use the + for manipulating strings in many other SmartBASIC commands. For example, the command:

```
PRINT a$+b$
```

is the same as:

```
PRINT a$;b$
```

This is a good way to remember that order is important when joining strings.

Often a program to manipulate words needs a string variable which has each letter in the upper- or lowercase alphabet. One type of program that can use these strings is an alphabetizing program. A good typist could create the string by entering:

```
LET alph$ = "abcdefghijklmnopqrstuvwxyz"
```

but it's easy to make mistakes and not fun to fix them. This is the perfect kind of task for ADAM—one where accuracy is important but thinking is not needed. We will define *alph*$ using the ASCII codes, the CHR$ command (see chapter 8) together with the +. All that is needed is to remember that the ASCII codes for, say, the lowercase letters run from 97 to 122 (see Appendix A). The following program fragment works quite nicely:

```
alph$ = ""
FOR q + 97 TO 122
LET alph$ = alph$ + CHR$(q)
NEXT q
```

This fragment gives the string variable *alph*$ the value equal to the complete lowercase alphabet. If the fragment is placed near the beginning of the program, *alph*$ can be used almost anywhere in the program—and you can be sure that no mistakes were made in typing. (Of course, be careful that no other string variable name in the program begins with the two letters *al*.)

When we first mentioned strings, we said that SmartBASIC allows string variables to have at most 255 characters. We didn't find this out by reading it in a book somewhere; instead, we discovered it using a short test program. (You can design your own test programs to check ADAM's various capabilities and limitations. You may find out things about ADAM no one else knows—not even its designers.) Our test program keeps making a string variable longer, and prints out the length at each stage. Eventually ADAM gets overwhelmed and prints the error message:

```
String Too Long in (some line)
```

At that point, ADAM's limit has been reached. Here's the program:

```
10 REM string length?
20 LET test$ = "": LET c = 0
30 LET c = c + 1
40 LET test$ = test$ + "x"
50 PRINT c
60 GOTO 30
70 END
```

This program traps ADAM in an infinite loop and never gets to line 70. It must (and does) end by forcing an error. After printing the error message, ADAM stops the program. The last number that appears on the screen before the error message is the maximum length of a string.

Forcing errors never hurts ADAM. This is why programs that push ADAM to its limits often use infinite loops which end by forcing an error. Of course, before using this technique be sure that an error will come out of the program. Otherwise you will have to stop these programs by using CONTROL/C or the RESET button. (Later in this chapter we'll show you programs that even CONTROL/C can't stop.)

Self-Check 1

A. Questions

1. If $a\$$ = "ADAM" and $b\$$ = "and EVE" then does $a\$+b\$$ = "ADAMand EVE" or "ADAM and EVE"?
2. How can you change the previous program fragment so that it creates a variable containing the uppercase alphabet?
3. What does PRINT "1" + "2" give on your screen?

B. Programs

1. Write a program (fragment) that defines a variable containing all the symbols on the keyboard.
2. Write a test program that shows that the value of an integer variable must be between −32767 and +32767.

Pretty Printing Revisited

Suppose you wanted to center a message on the screen (or on a page in the SmartWriter printer). Before we show you the general method, let's work through a simple example. If a message is ten characters long, then you must have five characters on either side of the center point of the line. For example, because a printed line has eighty possible positions, the half-way point is forty. To center a ten-character message, ask ADAM to SPC(35) (TAB may also work but, in older versions of SmartBASIC, it is unreliable for numbers bigger than thirty-one) and start printing the message from there. The general method is no harder:

1. Find the length of the message.

2. From the half-way point, backspace half the length of the message.

To write a module that performs these operations, we need a command to find the length of a string. In SmartBASIC it is:

```
LEN(     )
```

where the parentheses holds a string expression—that is, the name of a string or a string itself. When LEN( ) counts the length of a string; it counts blanks as well. So when:

```
a$ = "I'M ADAM"
```

ADAM will tell us (if we ask) that:

```
LEN(a$) = 8
```

If:

```
words = "Four score and seven. . ."
```

then:

```
LEN(words) = 23
```

because all the spaces and the three periods are counted.

Here's a subroutine to center a message (held in the variable or box labelled *c7*$ on the screen.

```
998 REM
999 REM ********CENTERING****************
1000 HOME
1010 LET m7 = LEN(c7$)
1020 IF m7 > 31 THEN PRINT "MESSAGE TOO LONG": GOTO 1060
1030 VTAB(12): HTAB(16-m7/2)
1040 INVERSE: PRINT c7$
1050 NORMAL
1060 RETURN
1061 REM ***********************************
```

Line 1030 tells ADAM to center the message. We have ADAM compute where to start printing and then use the HTAB command to position the cursor at that position. The variable *m7* holds the length of the string we want ADAM to print. It accomplishes two purposes. First it lessens the amount of typing needed to code the program. More important, a program that uses the length of a string many times runs faster when ADAM computes the length once and stores it as a numeric variable.

If you don't want to use INVERSE, just remove the line. An even better idea is to allow both normal and inverse printing of the centered string. The subroutine must be told (by having the program send a special kind of global variable, called a *key*, to the subroutine) which type of printing to use. (Recall that a global variable carries information to, from, or between subroutines.) The subroutine must then include lines that use the value of the key to control the display. For example, the following lines add this option to the subroutine:

```
1031 IF key = 1 THEN INVERSE
1032 IF key = 2 THEN FLASH
1040 PRINT c7$
```

Notice how a small change in the subroutine, together with feeding it a little more information, causes a large change in the way the string is displayed. This again shows the versatility of subroutines—a minor change in a few lines of code can make a major change in the way the program works.

This module is formatted differently from previous ones. The earlier modules have their names between rows of stars with several empty REM statements before the name. The way subroutines are formatted has no effect on the operation of a program. In this chapter, we'll show you a few other common ways of formatting modules. Use the one you like best or invent your own. The best format is the one that makes the program easiest to read, debug, and if necessary, improve in the future.

The LEN command can also be used to format the whole screen display. If a message has more than thirty-one characters, ADAM prints the first thirty-one on one line and then continues printing on the next line. (*Wrap-around* is the buzzword for this type of printing.) More often than not, part of a word is printed on one line and the remainder on the next. It's annoying to read a display with words split. Since ADAM can use the length command to check whether a word will be printed in pieces, we can tell ADAM to start printing that word on the following line. Many different subroutines can be used to force ADAM to print words in one piece. The first one, given below, is used when reading words from a DATA list and printing them using the SmartWriter. The command PR#1 must be turned on before the subroutine call.

```
1996 REM
1997 REM     ****************************
1998 REM     *  first formatting module  *
1999 REM     ****************************
2000 LET s7$ = "": REM initialize as an empty string
2010 READ a9$
2020 IF a9$ = "endword"  THEN PRINT s7$: GOTO 2070
2030 IF LEN(s7$ + " " +a9$) <= 80 THEN
s7$ = s7$ + " " +a9$: GOTO 2010
2040 PRINT s7$
2050 PRINT
```

(continued)

```
2060 s7$ = a9$: GOTO 2010
2070 RETURN
2071 REM ************************************
```

This subroutine uses several tricks, so it's worth spending some time on. ADAM must be reading some DATA list, otherwise line 2010 makes no sense. Line 2020 checks whether ADAM has reached the end of the data by testing if the string *a9$* it has just read is "endword". It is a good idea to put a comma after *endword* in the DATA list. Otherwise, there may be some spaces following it and the IF clause in line 2020 will always be false. (*endword* is often used as a delimiter. Some variants of it are *end-word, endofdata* and *eod*. All these delimiters wave a flag to tell ADAM that there is no more data in the list. Something that carries information on the status of ADAM, a DATA list, or a file, is called a *flag*.)

When ADAM reaches the DATA entry *endword,* whatever has been saved in the variable *s7$* is printed and control passes out of the subroutine (line 2070). Otherwise ADAM processes line 2030. That line checks that the string—obtained by concatenating *s7$* with a space and the new word *a9$*—has no more than eighty characters. If this is true, the strings are joined and ADAM goes back for more. When the string is too long, ADAM processes the group of commands beginning on line 2040. Lines 2040–2060 do the following:

1. Print what ADAM has been saving.

2. Return the carriage (line 2050).

3. Go back and re-initialize the string variable *s7$* so that it is assigned the word that would have been cut into pieces.

4. Begin formatting the next line.

(To use this subroutine in a more general program, replace the 80 on line 2030 by a global variable. When the value of this variable is 31, the subroutine formats the screen display. When the value is 80, it formats the printed text.)

If you are only working with screen displays, SmartBASIC has a command that can simplify this subroutine. The function POS(0) gives the current position of the cursor. The first position on a screen line is numbered 0 and the last has number 30.

In direct mode, if you enter:

```
PRINT "HELLO";:PRINT POS(0)
```

ADAM responds:

```
HELLO5
```

(We find it easiest to remember what POS(0) does by thinking of it in the following way: POS(0) tells how many characters have been printed starting at the beginning of the line. ADAM printed five characters in the word *HELLO*, so POS(0) is 5.)

POS(0) can be used in a line like:

```
IF LEN(a$) + POS(0) > 30 THEN PRINT "The next
word will drive me to a new line"
```

The IF clause checks whether the cursor will go past the thirty-first position if the next word, *a*$, and a space were to be printed. (The number 30 is used instead of 31 to account for the spaces between the words.)

You can also find out which screen line the cursor is on by using the function:

```
VPOS(0)
```

When using this command remember the top of the screen is numbered 0 and the bottom line is 23. (We keep this straight by thinking that VPOS tells how many complete lines have already been printed on the screen.)

In direct mode, the command:

```
PRINT VPOS(0)
```

tells you the position of the cursor before the command was processed. (The PRINT statement moves the cursor down one line.) In a program, you can assign the value of VPOS(0) to some variable, for example:

```
vp = VPOS(0)
```

VPOS can test if some information will scroll off the top of the screen when another PRINT command is processed. If VPOS(0) is 23, the next line printed forces the display to scroll.

Self-Check 2

A. Questions

1. What is the value of LEN("MADAM I'M ADAM")?
2. How would you change program 13.2 so that it centers printed text?
3. If you run the following program, what will appear on the screen?

```
10 HOME
20 PRINT "HELLO"
30 PRINT:PRINT
40 PRINT "GOODBYE";
50 PRINT POS(0), VPOS(0)
60 END
```

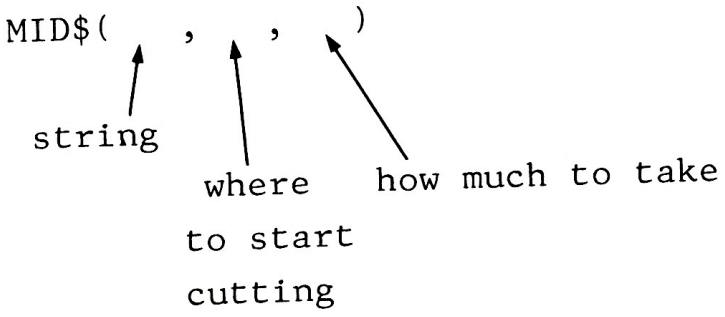
B. Programs

1. Write a subroutine using POS that simplifies the program on page 320.
2. Write a program that does the following things when printing on the screen:

1. Keeps count of the number of lines.
2. Makes sure that no word is cut into pieces.
3. When a display uses fewer than 24 lines, it is centered on the screen.

Taking Words Apart

The first command we showed you in this chapter joined two strings together to form a new string. Even more important are the commands in BASIC to cut up strings. Individual letters can be pulled out or whole pieces can be chopped out of a string to get new strings. The most important of the commands for chopping up strings is:



The first entry inside the parentheses holds the string (or name of the string) you want to cut up. Next comes the position of the first character that you want cut out of the string and finally the number of characters you want ADAM to pull out in a row. Here are some examples. Suppose:

```
a$ = MID$("ADAM and EVE", 1,2)
```

then *a*$ = "AD". ADAM has started at the first character and pulled out two characters—the character in the starting position and the next one.

We get the same result by using the variable:

```
b$ = "ADAM and EVE"
```

and then entering:

```
a$ = MID$(b$,1,2)
```

Here are some other examples using the same *b*$. If:

```
c$ = MID$(b$,4,5)
```

then:

```
c$ = "M and"
```

since we are starting at the fourth character and pulling out a total of five characters (including spaces).

If you leave out the last entry—the one telling ADAM how many letters to pull out—ADAM will retrieve the rest of the string from that position on. So:

```
MID$(b$,4) = "M and EVE"
```

and

```
MID$(b$,3) = "AM and EVE"
```

MID$ is one of the most useful commands in BASIC. Most of this chapter is devoted to its use.

Suppose we want to "explode" a word by printing it vertically instead of horizontally. To do this each letter must be pulled out of the word in the correct order and printed on a different line. ADAM has to do one operation repeatedly. It has to grab a letter from the word, print it, then move on to the next letter. These operations have to be performed as many times as there are letters in the word. Obviously, this calls for a FOR-NEXT loop. Here's the program:

```
10 REM MID$ demo
20 INPUT "Enter a word? ";q$
30 FOR i = 1 TO LEN(q$)
40 PRINT MID$(q$,i,1)
50 NEXT
60 END
```

All the action in this program occurs on line 40. On the first pass, $i = 1$, so ADAM starts pulling letters out from the first position. Since the last entry in MID$ is 1, ADAM pulls out only one letter. This gives the first letter in the word. On the next pass, $i = 2$, so ADAM starts at the second position and again pulls out one character from there (the last entry is still 1). Then ADAM prints the second letter of the word. ADAM continues in this way until it finishes pulling out all the letters. (If we put a semicolon at the end of line 40, the program works in a completely different way. It would print q in a rather silly and time-consuming way.)

As we said, changing this program slightly makes it behave very differently. For example, if you change line 30 to read:

```
30 FOR i = LEN(q$) TO 1 STEP -1
```

the program will explode the word backwards (last letter first). If you also add a semicolon to the end of line 40, ADAM will print the word backwards. We will use this trick in our *BOLDFACE* program (at the end of this section).

If line 40 were changed to:

```
40 PRINT MID$(q$,1,i)
```

then on each pass ADAM will pull out more and more characters. (The third position controls how many characters are pulled out and it increases on each pass through the loop.) However, the second entry (which controls where to start) stays the same—it is always the number 1. This means ADAM will always start from the first letter. At the last pass of the loop it prints:

```
MID$(q$,1,LEN(q$))
```

which prints the whole word. The end result is that this program gives a word ladder. For example, if *q*$ were "San Francisco", the program would print:

```
S
Sa
San
San
San F
San Fr
San Fra
.
.
.
San Francisco
```

The MID$ command can also simplify and improve some programs from earlier chapters. For example, here is another way to write a program to print random four-letter combinations (see program on page 214). We first set up a variable, *alph*$, containing the lowercase alphabet; then we will have ADAM take four random letters from our alphabet using MID$.

```
10 REM another way to get random four letter combinations
20 FOR q = 97 TO 122: alph$ = alph$ + CHR$(q): NEXT q
30 FOR i = 1 TO 4
40 let a = INT(26*RND(1) + 1)
50 PRINT MID$(alph$,a,1);
60 NEXT i
70 PRINT
80 GOTO 30
```

The crucial part of this program is the loop on lines 40–60. ADAM first finds a random integer between 1 and 26. Then we print the corresponding letter. For example, MID$ (*alph*$,7,1) is the seventh letter of the alphabet. The ; in line 50 tells ADAM to print the letters next to each other. To move the cursor to the next line before the loop begins again, there is an empty PRINT statement. Note that this program can only be stopped by CONTROL/C.

By setting up another variable, *vowel*$ = "aeiou", it's not hard to change this program so that at least one letter is a vowel chosen at random. This makes it more likely that the random four-letter combination will resemble a word. You might want to rewrite the program to force one letter to be a vowel—it can be done in many different ways.

Finally, let's write a BOLDFACE program. The key to printing in boldface is to send the printhead running backwards. As we said in chapter 8, the command PRINT CHR$(14) tells the printer to run backwards. In that chapter we could only print in boldface words like *madam* that read the same backwards as forwards. The program given here will boldface any string. There are several different ways to do this; the fastest (and best for ADAM) is to print a whole line before sending the printhead running backwards. After ADAM finishes a line we have to send the printhead running forward again. CHR$(15) will do it. Finally, since a line on the printer can only be eighty characters long, we have to cut apart the string if it has more than eighty characters.

(In the early versions of SmartBASIC, ADAM kept track of the number of characters printed regardless of direction. After eighty characters have been printed—either backwards or forwards—ADAM automatically forces a carriage return. This is supposed to be changed in the newest versions so the carriage will automatically return only after it moves forward eighty positions. Until the new version is released, this program can only print in boldface strings having no more than forty characters. To make it work with longer strings, change the subroutine on lines 1000–1060 so that all the 80s are 40.)

This program seems to work better with some printers than with others. It works fine when your printer is aligned well. On some SmartWriters, certain letters line up so poorly that the program can't work. You might want to call Coleco's 800 number for information on realigning the printer.

Here's the program:

```
10 REM a boldface program
20 HOME
30 VTAB(12): INVERSE
40 PRINT "Type whatever you want to boldface,"
50 PRINT "then hit return."
60 NORMAL
70 INPUT st$
80 wd$ = st$: GOSUB 1000
90 GOSUB 1500
100 END
995 REM
996 REM
997 REM ************************************************
998 REM       separate into lines
999 REM ************************************************
1000 nw = INT(LEN(wd$)/80)
1010 DIM bw$(nw)
1020 FOR q = 0 TO nw -1
1030 bw$(q) = MID$(wd$,1+q*80,80)
1040 NEXT q
1050 bw$(nw) = MID$(wd$,1+nw*80)
1060 RETURN
1495 REM
1496 REM
1497 REM ***************************************
1498 REM  BOLDFACE
1499 REM ***************************************
1500 PR#1
1510 FOR q = 0 TO nw
1520 PRINT bw$(q); CHR$(14); " ";
1530 FOR r = LEN(bw$(q)) TO 1 STEP -1
1540 PRINT MID$(bw$(q),r,1);
1550 NEXT r
1560 PRINT CHR$(15)
1570 NEXT q
1580 PR#0
1590 RETURN
```

We've set up an array, *bw*$( ), to hold the separate lines. Each line that we want ADAM to print becomes a separate entry in the list *bw*$.

After taking the trouble to print text in boldface, you usually want it to look as good as possible. For example, you wouldn't want a word to be chopped in two because ADAM returned the carriage in the middle of printing the word. Another subroutine can rearrange the entries in *bw*$ so that no word is cut into two pieces. You'll see the rearrangement technique in the next section.

Self-Check 3

A. Questions

Suppose *test*$ = "I am a friendly computer". What is:

1. MID$(*test*$,1,4)?
2. MID$(*test*$,3,5)?
3. MID$(*test*$,1,LEN(*test*$))?
4. MID$(*test*$,4)?
5. MID$(*test*$,1)?

B. Programs

1. Write a program to print every other word in a string.
2. Write a program to mix up a word by interchanging every two letters.
3. Write a program that uses a word to make a reverse ladder (starting with the whole word and printing smaller and smaller pieces).
4. Modify the random four-letter combinations program to make sure each word has at least one vowel.

More on MID$

Suppose we want to write a subroutine that counts the number of words in a sentence. The outline is pretty simple: tell ADAM to count the number of spaces between words and stop when it finds a period. We will use the MID$ command to pull out every character in the sentence. ADAM then checks whether the character is a space; if it is, ADAM adds one to a counter variable. If the character is a period, one more is added to the counter and the subroutine ends. *To avoid complications in this program and the ones that follow it, we only permit sentences that end with a period.* If you want to allow other punctuation you will have to add more IF-THEN tests.

```
997  REM ********* Word count ****************************
998  REM st$ will come from outside
999  REM ************************************************
1000 ct = 0
1010 FOR i7 = 1 TO LEN(st$)
1020 LET tmp$ = MID$(st$,i7,1)
1030 IF tmp$ = " " THEN ct = ct + 1
1040 IF tmp$ = "."  THEN ct = ct + 1: GOTO 1060
1050 NEXT i7
1060 RETURN
```

We can modify this subroutine so that it also counts the number of times a specific word appears. The easiest way to search for a word is to set up a new variable, *tw*$ (for temporary word). This variable starts out blank. Each time the subroutine finds a space, the characters up to the next space (or the period) are stored in *tw*$. Then *tw*$ is compared with the word we're searching for. If the temporary word (*tw*$) is equal to the test word (*tst*$—the word we're searching for), we have ADAM increase a different counter, *kt*, by 1. Here's the new program:

```
990 REM
991 REM
992 REM  ************************************************
993 REM  word count and word search subroutine           *
994 REM ************************************************
995 REM tst$ is the word searched for.      *
996 REM tst$ and st$ will come from outside *
997 REM ************************************
998 REM
999 REM ******WORD SEPARATION***********
1000 ct = 0 : kt = 0 : REM kt counts the number of
appearances of the word tst$
1010 twd$ = "": REM initialize
1020 FOR i7 = 1 TO LEN(st$)
1030 LET tmp$ = MID$(st$,i7,1)
1040 IF tmp$ = "." THEN GOSUB 1500: GOTO 1080
1050 IF tmp$ = " " THEN GOSUB 1500
1060 IF tmp$ <> " " THEN twd$ = twd$ + tmp$
1070 NEXT i7
1080 RETURN
1090 REM ************************************
1100 REM
1110 REM
1499 REM ********WORD SEARCH******************
1500 IF twd$ = tst$ THEN kt = kt + 1
1520 ct = ct + 1
1530 twd$ ="" : REM reinitialize
1540 RETURN
1541 REM ****************************************
```

Notice that the small subroutine, which begins on line 1500, increases the counters *and* reinitializes the variable *twd*$. After reinitializing *twd*$, ADAM can go back and check the next word. We did this here rather than in the original subroutine because, as we mentioned in chapter 12, you shouldn't load too many clauses onto an IF-THEN. Without reinitializing and doing the various counts in this (sub)subroutine, line 1030 would have had four clauses—and that's a bit much.

You may have wondered how ADAM's built-in word processing program can search for and then replace words. A good part of this task is accomplished by a program built into ADAM that uses the same logic as the two subroutines we just showed you. We think it's fun to add the necessary lines to the "word-search" subroutine to allow word replacement. In principle, replacement is simple. After you find the word, you just replace it. However, because strings in SmartBASIC can only be 255 characters long, we have to be a little careful. Before replacing the word, ADAM has to check that the new string (including the new word) isn't too large. Let's forget about this subtlety for a second and just concentrate on how to print a string with certain words changed. One way makes just a small change in the word-search subroutine.

```
1499 REM ****WORD SEARCH REVISITED*********
1500 IF twd$ = oldwd$ THEN PRINT newd$;
1510 IF twd$ <> oldwd$ THEN PRINT twd$;
1520 PRINT " ";
1530 twd$ ="": REM reinitialize
1540 RETURN
1541 REM ****************************************
```

The replacement word (*newd*$), the word replaced (*oldwd*$), and the string (*st*$) must come from outside the subroutine—perhaps from INPUT statements.

To do anything other than print the new word instead of the old, you have to do things differently. One way to get around the restrictions on the size of strings is to break the string up into individual words and store them in a list. After doing this it's easy to replace a word or even a group of words—just change the entries in the list. Here's a subroutine that changes a string into a list of words.

```
1996 REM ********************************
1997 REM Change a string into an array of words
1998 REM st$ comes from outside
```

(continued)

```
1999 REM **********************************
2000 twd$ = "" : REM initialize the test word
2010 DIM sw$(5000): REM empty string lists don't use a lot of
space
2020 wi = 1
2030 FOR i8 = 1 TO LEN(st$)
2040 LET tmp$ = MID$(st$,i8,1)
2050 IF tmp$ = " " THEN sw$(wi) = twd$: twd$ = "":
wi= wi + 1: GOTO 2070
2060 IF tmp$ = "." THEN sw$(wi) = twd$: GOTO 2080
2070 twd$ = twd$ + tmp$
2080 NEXT i8
2090 RETURN
2100 REM **********************************
```

We dimension the string list *sw*$ at 5000 to allow use of this subroutine again. If you had a long text, you could repeatedly call the subroutine. We'll leave these changes to you. It may seem that we've wasted a lot of space by dimensioning the string list so large, but we haven't. Recall, from chapter 11, that empty string lists use comparatively small amounts of ADAM's memory.

Although the previous program sidesteps using the obvious idea of simply inserting the string, it is still worth learning how to slip (or remove) a sub-string into (or from) a string. (*Sub-string* is, of course, the buzzword for a string that's part of a bigger string).

Suppose we want to write a "jumble" program. A jumble is a type of puzzle that is often found in newspapers. You get a bunch of letters and are supposed to rearrange them so they form a word. A jumble program needs a long list of words. It then calls a subroutine that jumbles a randomly chosen word from the list. After that, it asks the person running the program to guess the word. ADAM can check whether a guess is correct with an IF-THEN that tests whether two strings are equal. The hard part of the program is the subroutine that jumbles the word. An outline is simple: randomly pull letters from the original word and reassemble them. Actually asking ADAM to do this is a bit tricky. Each time a letter is chosen, it must be removed from the word that is being jumbled (otherwise it might be used again).

Thinking of how to accomplish a similar task in real life often helps to clarify the problem at hand. One analogous situation is removing a damaged link from a chain. First you must find the link you want to change. Remove the piece of the chain up to that link and put it aside. Then put the damaged link in a different place. Put the remaining piece of chain in a third

```
1060 LET i3 = INT(ct*RND(1)+1)
1070 RESTORE
1080 FOR q = 0 TO i3
1090 READ a4$
1100 NEXT q
1110 RETURN
1495 REM
1496 REM
1497 REM ******************************************************
1498 REM       jumble the word
1499 REM ******************************************************
1500 jmb$ ="": b4$ = a4$
1510 m4 = LEN(b4$)
1520 IF m4 = 0 THEN 1590
1530 LET s4 = INT(m4*RND(1)) + 1
1540 jmb$ = jmb$ + MID$(b4$,s4,1)
1550 IF s4 = 1 THEN b4$ = MID$(b4$,2): GOTO 1580
1560 IF s4 = m4 THEN b4$ = MID$(b4$,1,m4-1): GOTO 1580
1570 b4$ = MID$(b4$,1,s4-1) + MID$(b4$,s4+1)
1580 GOTO 1510
1590 RETURN
1995 REM
1996 REM
1997 REM ************************************
1998 REM       DISPLAY THE WORD
1999 REM ************************************
2000 HOME
2010 VTAB(12)
2020 PRINT "Can you guess what word I've"; SPC(3);
" jumbled? "
2030 PRINT: PRINT
2040 INVERSE
2050 PRINT jmb$
2060 NORMAL
2070 PRINT "I'll give you only 30 seconds"; SPC(2);
"before I erase it"
2080 tim = 30: GOSUB 2500
2090 VTAB (12)
2100 INPUT "What is your guess? "; gu$
2110 IF gu$ < > a4$ THEN PRINT "Sorry that's wrong":
tim = 10: GOSUB 2500
2120 IF gu$ = a4$ THEN PRINT "Congratulations!":
tim = 10: GOSUB 2500
2130 RETURN
2497 REM
2498 REM
2499 REM *********TIMING LOOP******************
2500 FOR i = 1 TO tim
2510 FOR p = 1 TO 1000
2520 NEXT p
2530 NEXT i
2540 HOME
```

(continued)

```
2550 RETURN
4998 REM
4999 REM
5000 DATA clear, continue, delete, list, new, run, then,
error, return, resume, trace, call, memory, data, define
5010 DATA function, end, read, write, stop, next, nest, let,
restore, input, print, space, character, string, color
5020 DATA inverse, normal, plot, position, cursor, scale,
sort, screen, display, speed, backspace, bell, physical
5030 DATA logical, processing, execution, system, number,
variable, looping, graphics, endword,
```

Notice that we use a global variable, *tim*, in the timing loop. This lets us get different pauses from one timing loop. You can change line 2080 to allow more or less time to guess the jumble. Other ways to improve this program are to make the DATA list longer and add a subroutine to keep score.

Finally, there are two other variants on the MID$ command. To be quite honest they are really just artifacts. (*Artifact,* when used as a buzzword, means a command that has survived from some older programming language. It is not complementary.) The two commands are LEFT$ and RIGHT$.

As the names suggest, LEFT$ picks out characters from the beginning of a word and RIGHT$ picks them out from the end of the word. The number in the second position tells ADAM how many characters to pull out. Since MID$ can do all this and more, we rarely use LEFT$ or RIGHT$. Of the two, RIGHT$ is used more frequently. It avoids having to do subtraction in the MID$ command. For example, if you want the last seven characters of the string *q*$, you can give ADAM either of these commands:

```
MID$(q$,len(q$) - 7)          RIGHT$(q$,7)
```

The second is a bit simpler and so is occasionally used. LEFT$ works the same way but doesn't save much work. If you want the first seven characters of a string, tell ADAM either:

```
MID$(q$,1,7)                  LEFT$(q$,7)
```

Some people like learning all the possible commands in a programming language. We prefer not to clog our heads with lots of commands—we'd rather spend the time learning to use the useful ones.

Self-Check 4

A. Questions

1. How would you remove the fourth letter of a word and leave the rest of the word unchanged?
2. What is LEFT$("HELLO",2)?
3. What is RIGHT$("GOODBYE",3)?

B. Programs (these are quite involved)

1. Write a program to play hangman.
2. If you know pig latin you now have enough commands to write a program to translate English into pig latin. Try it.

Some Advanced Features of Strings

Suppose you had a string and wanted to change certain capital letters to lowercase (or vice-versa). With the commands we've shown you so far it's possible but hard. The logic is simple; check each letter and change it if necessary. However, actually programming this would be extremely time-consuming. The only trick we can think of is to set up two string variables, one with the lowercase alphabet and the other with the uppercase. Then tell ADAM to exchange letters as needed. Although this might simplify the program slightly the program would remain extremely clumsy.

The program we'll show you below uses a nice feature of the ASCII codes for the uppercase and lowercase alphabets. If you look at the table of ASCII codes in Appendix A, you'll see that the code numbers for the upper-

case alphabet range from 65 to 90, and for the lowercase alphabet the code numbers range from 97 to 122. The letter-changing subroutine has three parts:

1. Get the ASCII code for the letter.
2. Change the ASCII code.
3. Change the new code to the character it represents.

The changes in step 2 are simple:

(To go from caps to lowercase)	Add 32 to the ASCII code of each letter.
(To go from lowercase to caps)	Subtract 32 from the ASCII code of each letter.

The command that gives the ASCII code for any character or symbol is easy to remember. It is:

```
ASC(    )
```

Here are some examples:

```
ASC("A") = 65
ASC("!") = 33
```

The ASC( ) command only works with strings. If you try it with numbers you'll get the error message:

```
Type Mismatch
```

Also if the string is longer than one character, ASC( ) only gives the code for the first character and disregards the remaining characters.

Here's a program that changes every uppercase letter in a string to the corresponding lowercase letter.

```
10 REM a lower case converter
20 INPUT "Enter some string. "; wd$
30 FOR i = 1 TO LEN(wd$)
40 x$ = MID$(wd$,i,1)
50 IF ASC(x$) > 64 AND ASC(x$) < 91 THEN
x$ = CHR$(ASC(x$)+ 32)
60 PRINT x$;
70 NEXT i
80 END
```

Chapter 10 contains a technique (program at bottom of page 240) for alphabetizing a list of words. We mentioned that it has a major flaw, namely, it does not work correctly if the words have both upper- and lowercase letters. Using the ASC command and the MID$ command lets us modify that program to work in all cases. First, here's that program written as a subroutine:

```
1997 REM ************************************************
1998 REM  a sorting subroutine
1999 REM ************************************************
2000 FOR j = 1 TO n-1
2010 FOR i = j + 1 TO n
2020 IF x$(i) >= x$(j) THEN 2060
2030 t$ = x$(j)
2040 x$(j) = x$(i)
2050 x$(i) = t$
2060 NEXT i
2070 NEXT j
2080 RETURN
2090 REM ******END SORT*************************************
```

The key to making this sorting routine work for strings is to use a more complicated comparison than the one on line 2020. We have to overrule ADAM's normal way of comparing strings. We will send ADAM to a subroutine that compares the strings character by character and disregards the differences between upper- and lowercase letters. Line 2020 becomes a GOSUB to this subroutine.

```
2020 w1$ = x$(i) : w2$ = x$(j): GOSUB 2500
```

We also use a global variable named *key*, controlled by the subroutine, that tells ADAM whether to switch the strings. If *key* = 0, ADAM does nothing; if *key* = 1, ADAM switches the strings.

```
2497 REM ************************************************
2498 REM    STRING COMPARISON REVISITED
2499 REM ************************************************
2500 t1 = LEN(w1$): t2 = LEN(w2$): key = 0
2510 IF t2 < t1 THEN t1  = t2: REM find smaller word
2520 FOR i1 = 1 TO t1
2530 a1$ = MID$(w1$,i1,1): a1 = ASC(a1$): b1$ =
MID$(w2$,i1,1): b1 = ASC(b1$)
2540 IF a1 > 95 AND a1 < 127 THEN a1 = a1 - 32
2550 IF b1 > 95 AND b1 < 127 THEN b1 = b1 - 32
2560 IF a1$ < b1$ THEN key = 1: GOTO 2580
2570 NEXT
2580 RETURN
```

To take advantage of the key we must add (to the original subroutine):

```
2025 IF KEY = 0 THEN 2060
```

The second submodule is worth discussing. The strings we want to test are contained in *w1*$ and *w2*$. Instead of the old line 2020, we have to use a character-by-character comparison. To do this, we first have ADAM find the shorter of the two words. If you forget this, ADAM will try to compare too many characters and give an error message. Next, ADAM strips off one character from each word and converts each one into the ASCII code for the uppercase letter. Now ADAM uses its ability to compare numbers. As long as each character of *w1*$ comes after the corresponding character of *w2*$, the subroutine keeps the value of the key at 0. If one comes before, then the subroutine changes the value of the key to 1 and returns.

Another way to use the ASC( ) command is to verify the codes given for each key (or key combination) in Appendix A. When we were checking this our first instinct was to use a program like:

```
10 REM ASCII codes
20 PRINT "press a key"
30 INPUT "then hit ENTER ";a$
40 PRINT "The ASCII code for "; a$ ;"is ";ASC(a$)
50 GOTO 20
```

We got tired of having to hit ENTER each time the **?** appeared. Besides the fact that this program can only be stopped by CONTROL/C, it doesn't work for some of the keys. Luckily, SmartBASIC has a command called GET that grabs individual characters from the keyboard *without* requiring that you hit ENTER. The GET command is deceptively simple. If you enter:

```
GET a$
```

ADAM places the next character hit on the keyboard in the box labelled *a*$. You won't have to hit RETURN or do anything else. This sounds so simple you might be tempted to forget the INPUT command and only use GET with concatenation. Well it is, *but* . . . no prompt is shown and, unless you use a PRINT statement to show one, you will never know ADAM is sitting and waiting for a key to be pressed.

> THE COMMAND GET (*name of a string variable*) SUSPENDS THE PROCESSING OF ANY PROGRAM UNTIL A KEY IS PRESSED. THE FIRST AND ONLY THE FIRST KEY (OR KEY COMBINATION) PRESSED BECOMES THE VALUE ASSIGNED TO THE VARIABLE.

ADAM allows you to enter a number in the same way. For example, just enter:

```
GET a
```

instead of GET *a*$. The variable *a* is assigned whatever number is hit. Since only one key is recognized by a GET command, only numbers between 0 and 9 can be entered this way. However, if you use this version of the GET command and press a letter key, ADAM responds with "Syntax Error" and your program bombs. More generally, you can use:

```
GET  IT varnm IT
```

where *varnm* is the name of some numeric variable.

The GET command can make the tests often used to start a program more elegant. Instead of:

```
INPUT "Enter y or n ";ynS
IF yn$ = "y" OR yn$ ="Y" THEN  IT something IT
```

which requires hitting the return key, you can use:

```
PRINT "HIT the y key to continue"
GET yn$
IF yn$ = "y" OR yn$ = "Y" THEN (something)
```

Since beginners may be confused by the word *enter,* the GET command may be easier for them.

Another elegant use of the GET command is to end a graphics program with the statements:

```
PRINT "Press any key to end"
GET a$:TEXT
END
```

The *a$* is used as a dummy variable. Nothing happens until a key is pressed. After ADAM gets the dummy variable it executes the TEXT command to erase the picture.

GET can be dangerous—it can hang your machine. (*Hang* is the buzzword for when a computer won't respond.) You can press keys—even CONTROL/C—and nothing happens. To hang your machine, all you need do is run the following program (DON'T DO IT!):

```
10 REM hanging ADAM
20 GET a$
30 GOTO 20
```

Suppose you try to break this infinite loop by pressing CONTROL/C. All ADAM does is assign CONTROL/C (ASCII code 3) to *a$*. The GET command overrides ADAM's normal way of handling CONTROL/C. The result is that ADAM has been hung.

If your computer ever hangs, don't despair. Just reload the SmartBASIC tape and start over. Unfortunately, any program in memory is lost, but be assured ADAM hasn't been hurt.

Here's the program we actually used to check the ASCII codes of the keys.

```
10 REM another ASCII search
20 PRINT "Press any key - CONTROL/C to stop"
30 GET a$
40 IF a$ = chr$(3) THEN 100: REM CONTROL/C test
50 PRINT "The ASCII code for"; a$; "is "; ASC(a$)
60 GOTO 20
100 END
```

Self-Check 5

A. Questions

1. What is the difference between the GET command and the INPUT command?
2. In the modification to program on the top of page 340 (line 2025), why did we only check that *key* = 0?

B. Program

Write a program that uses a loop and the GET command to input a name. Be careful to include a way to end the program.

STR$ and VAL

Because string manipulations are so powerful, all versions of BASIC give you a way to change numbers into strings. In SmartBASIC the command is:

```
STR$(       )
```

If, say, $x = 1234567$, and you enter the command $x\$ = STR\$(x)$, then:

```
x$ = "1234567"
```

At first it may not seem like this command is very useful. After all, when you change a number into a string you lose the ability to add, subtract, or multiply it. What you gain in return is the ability to isolate each digit. (Digits following the decimal point are called *decimal digits.* The first digit before the decimal point is the ones digit, the next one to the left is the tens digit, etc.)

One unfortunate use of the STR$ command is to target junk mail. Companies obtain lists of names, addresses, and zip codes. They also know (from marketing surveys) the zip codes for "up-scale" areas (which is advertising lingo for having money to spend). The junk mailers can check the last three digits of your zip code to see whether you should be a target for their advertisements. They might use a program with lines like:

```
READ zip
zip$ = STR$(zip)
IF MID$(zip$,3) = ... THEN
```

where the . . . are filled in by the last three digits of the areas they are targeting.

The STR$ command can also isolate each digit in a number. For example to explode a number from the highest digit down, use the following program:

```
10 REM     explode a number
20 INPUT "Enter some number"; n
30 n$ = STR$(n)
```

(continued)

```
40 FOR i = 1 TO LEN(n$)
50 PRINT MID$(n$,i,1)
60 NEXT
70 END
```

Warning: If you ask ADAM to explode a number smaller than .01 or larger than 999,999,999 using this program, you'll get something strange. ADAM thinks of these numbers in scientific notation (see chapter 3), so the results come out weird.

SmartBASIC has a command, VAL, that undoes STR$—it turns a string into a number. VAL only works with strings like "123". If you use VAL with a string that's not made up of digits, the program will behave really strangely—we don't advise it.

VAL and STR$ are often used together in the same program. First use STR$ to change a number into a string. Then you have ADAM play with the digits using MID$. Finally use VAL to turn the digits back into numbers.

One common use of VAL and STR$ together is to "prettyprint" various dollar amounts. When listing a lot of dollar amounts, you generally want to line up all the decimal points in the same column. Another common feature of such programs is that they insert enough stars to the left of the number so that no one can change the ledger without being obvious. Finally, you always want two places to show up after the decimal point—even if they are both zeros.

This last point can be dealt with in many ways. One way is to first multiply every amount by 100, and then insert the decimal point. For example, suppose we start with 81. Multiplying it by a 100 gives 8100. Inserting the decimal point and a dollar sign gives $81.00. If we start with some figure like $1.001, though, we want to disregard the tenth of a cent. This is easy using INT. However, as mentioned in chapter 9, INT sometimes works strangely. This problem can be avoided if we add a very small number before we use INT (see line 4010 below).

Before writing the subroutines to prepare a ledger, let's decide what material the subroutine should work with. We can make this program quite general by working with a list containing the ledger entries. We will have ADAM manipulate the list. The first manipulation multiplies all the entries in the list by 100. Next we have ADAM find the entry with the largest number of digits. This entry tells ADAM how many stars will be needed to print the remaining entries.

Here are the subroutines:

```
3995 REM ************************************************
3996 REM the list of amounts comes from outside
3997 REM the array will be called do(  ) and runs
3998 REM from 1 to d7. The output is a ledger
3999 REM ************************************************
4000 FOR i = 1 TO d7
4010 do(i) = INT(100*do(i) + .001)
4020 NEXT
4030 GOTO 5000
4040 REM ************************************************
5000 let md = 0: REM find the longest entry
5010 FOR i = 1 TO d7
5020 do$ = STR$(do(i))
5030 IF LEN(do$) > md THEN md = LEN(do$)
5040 NEXT
5050 FOR i = 1 TO d7
5060 do$ = STR$(do(i)): GOSUB 5500
5070 NEXT i
5080 RETURN
5495 REM
5496 REM
5497 REM ************************************************
5498 REM  subroutine to prettyprint a single line
5499 REM ************************************************
5500 IF LEN(do$) = 1 THEN GOSUB 5700: RETURN
5510 LET id = md - LEN(do$): REM find the number of *'s
5520 IF id = 0 THEN 5560
5530 FOR j1= 1 TO id
5540 PRINT "*";
5550 NEXT j1
5560 FOR j = 1 TO LEN(do$)
5570 IF  j = LEN(do$)-1 THEN PRINT ".";
5580 PRINT MID$(do$,j,1);
5590 NEXT j
5600 PRINT
5610 RETURN
5699 REM ******taking care of pennies**********************
5700 FOR j1 = 1 TO md-2: PRINT "*";: NEXT j1
5710 PRINT ".0" + do$
5720 RETURN
5999 REM ************************************************
```

The main subroutine uses a FOR-NEXT loop (lines 5010–5040) to find that entry in the new list which is the longest in length. The rest of this subroutine just controls the information sent to the second subroutine.

The second subroutine checks whether the entry has only one digit. If so, control is sent to a separate subroutine. Otherwise, the subroutine prints each entry in the correct form. It's not long but it is tricky. Since the techniques it uses come up time and time again, it's worth examining in some detail. We will outline the main points.

The first FOR-NEXT loop (lines 5530–5550) determines how many stars to print. Because FOR-NEXT loops always make at least one pass, we have to be careful. Line 5520 has ADAM test that *id* is greater than zero. If it is greater than zero, then the entry to be printed has fewer digits than the maximum—so we pack it with stars. After this loop is finished ADAM moves on to the next loop (lines 5560–5590). This loop starts printing the number from the leftmost entry. When ADAM reaches the second digit (from the right), it adds the decimal point. The RETURN on line 5610 sends ADAM back for the next entry—after it has processed a PRINT command. The PRINT command tells ADAM to move to the next line.

Self-Check 6

A. Questions

1. What is STR$(123.2)?
2. Is x = VAL("12") + VAL("34") a legal statement in SmartBASIC, and if so, what does x equal?

B. Program

Write a program to decide if a number reads the same backwards as forwards.

Summary

In this chapter you learned how to manipulate strings. We showed you the commands needed to break strings apart or put them together. We also showed you how to convert numbers into strings and strings into numbers.

The new commands were:

LEN( )	to find the length of a string (including spaces).
MID$(, ,)	to pull out a substring from a string. The first entry holds the string (or a string variable). The second entry holds the character at which ADAM starts pulling. The third holds the number of characters (in order) to be pulled.

MID$(,)	this variant of the MID$ command pulls out the remaining characters in a string starting with the character at the position given by the second entry.
LEFT$(,)	pulls out the number of characters indicated in the second slot from the front of the string (or string variable) in the first slot.
RIGHT$(,)	works the same as LEFT$ except that it takes from the end of the string.
GET variable	pulls a single character from the keyboard *without* requiring that the RETURN key be pressed and assigns it as the value of that variable.
POS(0)	gives the current horizontal position of the cursor.
VPOS(0)	gives the current vertical position of the cursor.
ASC( )	gives the ASCII code of the symbol inside the parentheses.
STR$( )	changes a number into a string.
VAL( )	changes a string consisting only of digits into a number.

BUZZwords

ARTIFACT

CONCATENATION

DIGIT

FLAG

HANG

SUB-STRING

WRAP-AROUND



High-Resolution Graphics

In this chapter we will introduce you to high-resolution (high-res) graphics. High-res graphics will let you draw spectacular pictures on the screen.

We'll show you the following commands:

HGR HGR2 HCOLOR HPLOT

You'll see how the paddle can send ADAM information to use in SmartBASIC programs. You'll do this with the command:

PDL( )

The Basics

Anyone who has looked at a newspaper photo closely has seen high-resolution graphics in action. If you look carefully it's easy to see that a newspaper photograph is made up of hundreds of little black dots. ADAM's high-resolution graphics work the same way. ADAM can draw up to 49,152 dots on a television screen! Moreover ADAM will color each of these 49,152 dots in almost any way you want. So ADAM's pictures can be quite intricate, and often quite beautiful. The dots that ADAM uses in high-resolution graphics are usually referred to as pixels. The buzzword *pixel* is the abbreviation for *picture element.* The detailed pictures ADAM uses in its arcade-quality games are also made using these pixels.

The letters and symbols ADAM displays on the screen are also made up of high-resolution pixels (dots). Each letter is drawn by coloring a certain number of pixels in a six by eight grid. For example, figure 14.1 shows how ADAM draws a lowercase g and an uppercase A.

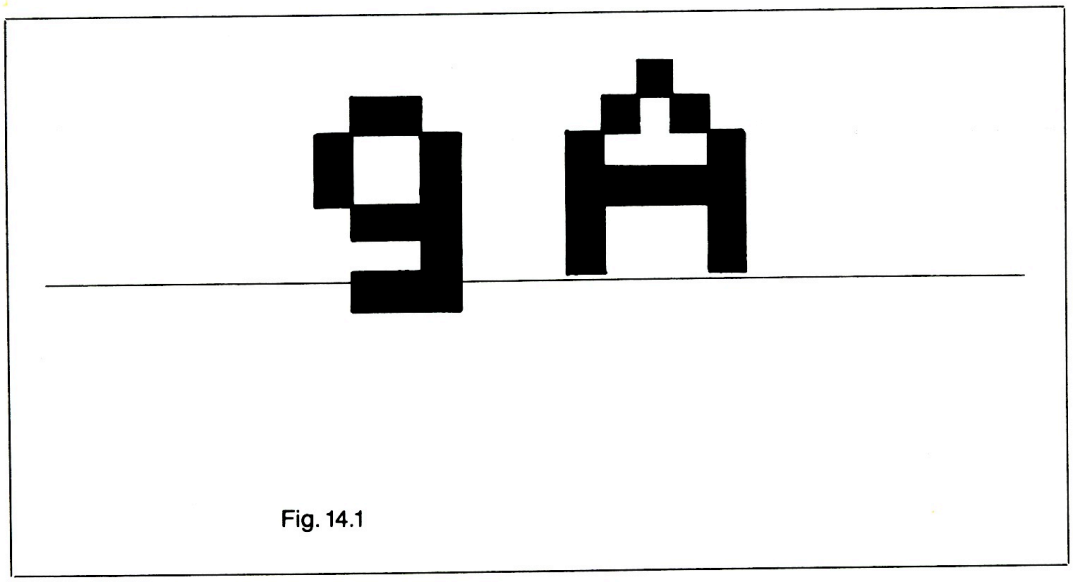


Fig. 14.1

The first seven rows give the part of the letter that lies above the writing line. The one row below the line displays what printers call descenders (for example the lower part of the g or p). Letters are always separated from each other by a space that is two pixels wide.

Before starting to work with high-resolution graphics, we suggest

sharpening the tools you picked up in low-resolution graphics (chapter 7). High-resolution programming techniques build on them. For example, table 14.1 lists the SmartBASIC commands in low-res along with their high-res counterparts.

Graphics Commands

low-res		**high-res**	
GR	(puts ADAM in low-res graphics mode)	HGR	(puts ADAM in high-res graphics mode)
COLOR	(tells ADAM what color to use)	HCOLOR	(tells ADAM what color to use)
PLOT	(ADAM colors in the block in the previously specified color)	HPLOT	(ADAM colors in the dot in the previously specified color)
TEXT	(takes ADAM out of graphics mode)	TEXT	(takes ADAM out of high-res graphics mode)

Table 14.1

In low-res graphics the screen has forty blocks running up and down and a four-line text window. After the command HGR, ADAM sets aside a grid 256 columns across and 159 rows down. The command HGR also leaves a four-line text window. In high-res graphics the positions for the pixels are numbered with the columns starting at 0 on the left and ending at 255 on the right. The rows are numbered from 0 (at the top) to 158 (right above the text window). Recall that the low-res graphics command PLOT must be followed by two numbers separated by a comma. These numbers tell ADAM where to plot a block. The first number is the column position and the second, the row. Its high-res analog, the HPLOT command, also needs the column position first and then the row. For example:

```
HPLOT 0,0          Top left-hand corner
```

HPLOT 255,0	Top right-hand corner
HPLOT 0,158	Lower left-hand corner (above the text window)
HPLOT 255,158	Lower right-hand corner (above the text window)

We often use grids to lay out low-res designs. They are less useful in high-res graphics. (Think about the thousands of different HPLOT commands needed to draw the dots for most sketches.) However, they can be helpful in designing line drawings. Figure 14.2 is a grid that you can photocopy.

Before you can do any plotting you'll need a list of ADAM's high-res colors.

High-Resolution Colors

0	Black	8	Dark Yellow
1	Green	9	Dark Blue
2	Dark Red	10	Grey
3	White	11	Light Red
4	Black	12	Dark Green
5	Medium Red	13	Light Yellow
6	Medium Blue	14	Sky Blue
7	White	15	Magenta

Table 14.2

Suppose you want to plot a blue dot in the center of the high-resolution screen. You might think you could use the following short program:

```
10 REM sorry it won't work
20 HGR
30 HCOLOR = 14
40 HPLOT 128,78
50 PRINT "Press any key to return to TEXT"
60 GET a$: TEXT
70 END
```

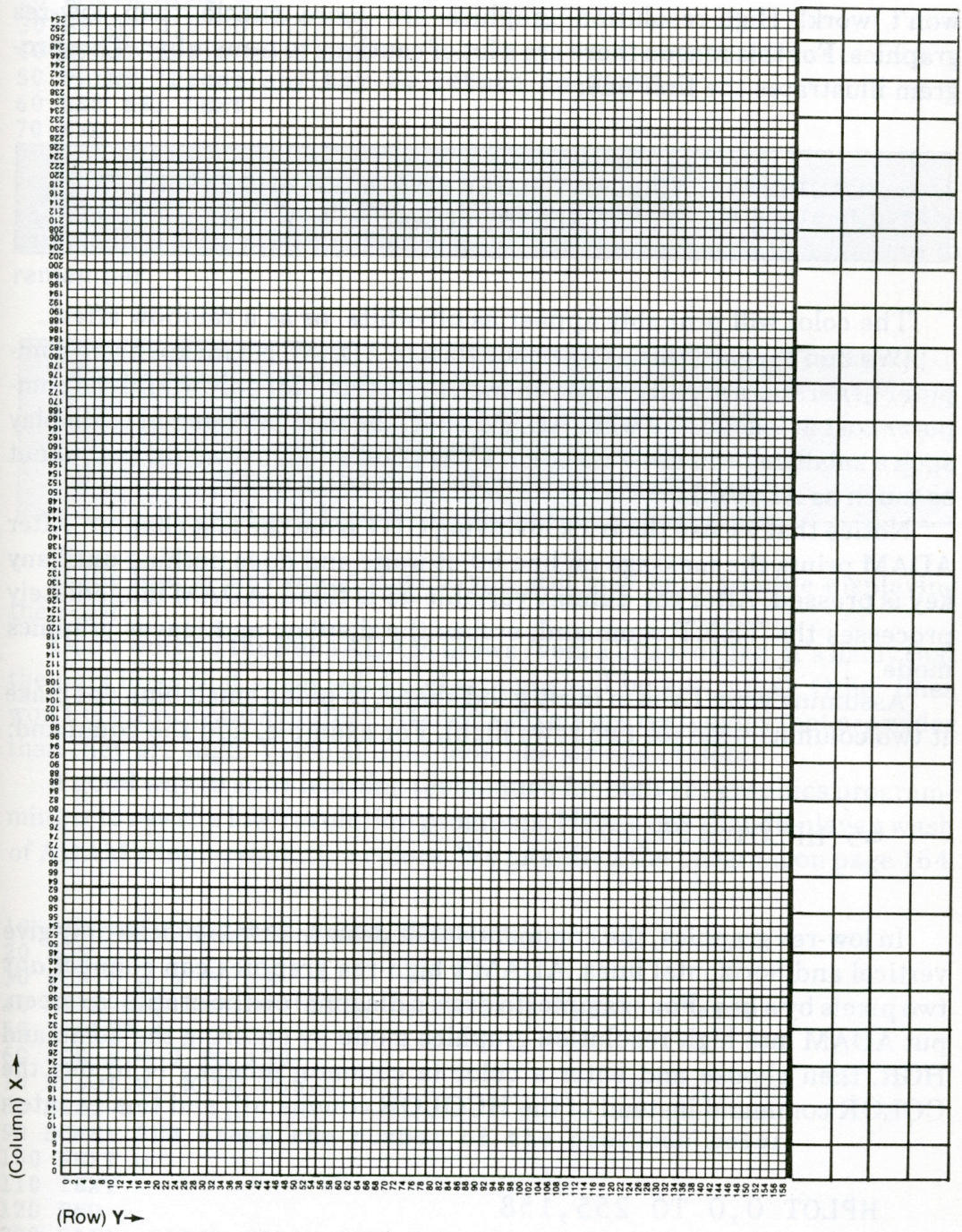


Fig. 14.2

However, as we said in the REM statement, this program probably won't work! High-resolution graphics are more subtle than low-res graphics. For this reason there are a lot of cautions to remember. This program illustrates the first one:

> CAUTION #1: YOU CAN'T PLOT A SINGLE DOT AND EXPECT TO SEE THE COLOR.

The color will probably appear as off-white, or as a rainbow effect.

(We don't regard this as a serious flaw in ADAM. No inexpensive computer generates a signal precise enough to do this. Not only would the computer cost at least twice what ADAM does, the monitor needed to display such a small colored dot—no color TV is good enough—would cost about as much as ADAM.)

Notice that we use the trick involving GET from the last chapter. After ADAM prints the message on line 50, it waits and does nothing until any key is pressed. After the dummy variable *a*$ is filled, ADAM immediately processes the TEXT command, erases the display, and leaves graphics mode.

Assuming your TV is in fairly good shape, to plot a small blue dot make it two columns wide (or two rows high). For example, add the command:

```
45 HPLOT 129,79
```

In low-res graphics, the commands that draw multiple blocks only give vertical and horizontal lines. ADAM's high-res graphics can connect *any* two pixels by a line. For example, to draw a diagonal on the graphics screen, put ADAM into high-resolution graphics mode by entering the command HGR, then choose and enter a color (a common mistake is to use the COLOR command instead of the HCOLOR command), and finally enter:

```
HPLOT 0,0 TO 255,158
```

To draw a giant red X you can use the program:

```
10 REM X marks the spot
20 HGR:HCOLOR = 2
30 HPLOT 0,0 TO 255,158
40 HPLOT 255,0 TO 0,158
50 PRINT "Press any key to return to TEXT"
60 GET a$: TEXT
70 END
```

To connect together points on ADAM's high-resolution screen, use the command TO in a HPLOT statement. Of course, there is a caution to remember.

> CAUTION #2: PLOTTING LINES NEXT TO EACH OTHER IN DIFFERENT COLORS WILL PROBABLY CAUSE BLEEDING (BLEEDING IS THE BUZZWORD FOR COLORS OOZING OVER ONE ANOTHER.)

If the line is only one dot wide you still may have trouble displaying the color.

Don't worry if the lines ADAM draws seem a little ragged. It's likely that the only lines that seem solid and straight will be horizontal. Other lines won't come out quite perfect. ADAM's pixels are rectangles that are wider than they are high. Vertical lines look like strings of dots.

A good way to start learning techniques for high-res graphics programming is to rewrite low-resolution programs. Here's one that displays a wash of ADAM's high-resolution colors. We modified the program on page 164.

```
10 REM an annotated color display
20 HGR
30 FOR c = 0 TO 15
40 READ ko$: PRINT "This is color # ";c;" - ";ko$
50 HCOLOR = c
60 FOR a = 0 TO 158
70 HPLOT 0, a TO 255,a
80 NEXT a
90 HOME: REM erase the number of the previous color
100 NEXT c
110 TEXT
120 END
200 DATA  BLACK,GREEN,DARK RED,WHITE1,BLACK2,MEDIUM RED,MEDIUM
BLUE,WHITE2,DARK BLUE,GREY,LIGHT RED,DARK GREEN,LIGHT
YELLOW,SKY BLUE,MAGENTA
```

It's also not hard to modify the low-res spectrum program (on the bottom of page 166). Since there are sixteen colors to display and 256 columns to do it in, we'll make each bar sixteen rows wide.

```
10 REM ADAM's high resolution spectrum
20 HGR
30 FOR ko = 0 TO 15
40 FOR i = 0 TO 15
50 HCOLOR = ko
60 HPLOT i + 16*ko,0 TO i+16*ko,158
70 NEXT i
80 NEXT ko
90 PRINT "PRESS any key to end"
100 GET a$: TEXT
110 END
```

The inner loop (lines 40–70) makes each bar 15 rows wide and the outer loop (lines 30 and 80) sets the color and positions the bar in the correct row. For example, on the first pass the inner loop tells ADAM to plot from row 0 to row 15. On the second pass *ko* = 1, so ADAM plots from row 16 to row 31, and so on.

When you run this program, the last bar may appear somewhat narrower than the other bars. If this happens it is a consequence of the overscan problem we talked about in chapter 7. You might want to run the overscan test given below. (It's a modification of the test in the program on page 163.)

```
10 REM an overscan search at the left edge
20 HGR: HCOLOR = 3: REM we use white for the test
30 FOR a = 0 TO 255
40 HPLOT a,0 TO a,158
50 PRINT "press y if you saw the line or any other key if you
didn't.  ":GET s$
60 IF s$ = "y" OR s$ = "Y" THEN 90
70 HOME
80 NEXT a
90 TEXT
100 PRINT "The first column you can use on the left is  "; a
110 END
```

Overscan is more obvious in high-resolution graphics, so don't be surprised if this test suggests starting in column 8 or 10. This is quite common and nothing to worry about. We've found that Sony televisions give good results and the cheaper black-and-white sets usually don't.

To make the other tests for overscan, use:

```
HPLOT 255-a,0 TO 255-a,158          (on the right)
```

For the top, change line 30 to read:

```
FOR a = 0 to 158
```

and change line 40 to:

```
HPLOT 0,a,TO 255,a
```

(If your TV has a severe overscan problem you'll have to modify the programs in this chapter to take it into account.)

Suppose we want to display a grid, like a chessboard, on ADAM's screen. A grid is made up of a bunch of horizontal and vertical lines. It's not hard to write a subroutine that draws a grid. To make the subroutine quite general, we will use a global variable named, say, *mesh*. In the program, this global variable tells ADAM how far apart to space the lines.

```
996 REM*******drawing a mesh************************
997 REM global variable - mesh (determines fineness)
998 REM local variables - i9,j9
999 REM ****************************************
1000 FOR i9 = 0 TO 255 STEP mesh
1010 fin = INT(158/mesh)*mesh
1020 HPLOT i9,0 TO i9,fin
1030 NEXT i9
1040 FOR j9 = 0 TO fin STEP mesh
1050 HPLOT 0,j9 TO i9 - mesh,j9
1060 NEXT
1070 RETURN
```

The only tricky point in this subroutine is the use of *i9 — mesh* and *fin*. These variables are used to make sure that the vertical lines don't extend beyond the horizontal ones and vice-versa.

As in low-resolution graphics, ADAM erases an entire picture after processing another HGR command. You can also erase any part of a picture by changing the HCOLOR to either of the black's (HCOLOR = 0 or 4) and replotting the picture.

Self-Check 1

A. Questions

1. What is the resolution of ADAM's HGR screen?
2. Does your television have an overscan problem in high-res graphics? If so, how much of the picture do you lose?
3. What would you do to modify the program on the top of page 355 so that it draws a blue X? Can you make it blink between red and blue? On and off?

B. Programs

1. Write a program that makes a larger A and g. Use ADAM's letter designs but display the letters in a larger format.
2. Write a program that draws stars on ADAM's HGR screen. How could you make the stars twinkle?

More on HGR

In high-res graphics ADAM lets you do more than connect any two points. ADAM will connect as many pixels as you want. For example, drawing a triangle on a piece of paper requires connecting three points by lines. You can ask ADAM to draw a triangle on the screen. For example:

```
HPLOT 20,20 TO 150,80 TO 150,140 TO 20,20
```

This command tells ADAM to draw three lines. They go from the first point to the second, from the second to the third, and from the third back to the first.

Here's a program that draws lots of triangles:

```
10 REM       lots of triangles
20 HGR: HCOLOR = 3
30 INPUT "Enter the column and row for the
first corner?  "; xl,yl
40 INPUT "and for the second corner? "; x2,y2
50 INPUT "now the third corner? "; x3,y3
60 HPLOT xl,yl TO x2, y2 TO x3, y3 TO xl,yl
70 HOME
80 PRINT "Do you want to draw another triangle?  Hit y or n"
90 GET a$
100 IF a$ ="y" OR a$ = "Y" THEN 30
110 TEXT
120 END
```

This program should have an error trap to make sure that ADAM is only asked to plot points on the high-res screen. Asking ADAM to HPLOT off the screen will bomb the program. We'll leave the writing of this to you. You might also want to make this program easier for a beginner to use by printing more information about the points that can be plotted.

ADAM has one other way of using the HPLOT command. If ADAM encounters a line saying:

```
HPLOT TO kol,rw
```

ADAM connects the last pixel it colored to the pixel at position indicated by *kol* (column) and *rw* (row). If ADAM hasn't previously plotted any points, this command draws a line from the upper left-hand corner to the point positioned at *kol* and *rw*.

We already showed you how to use ADAM to draw bar graphs. There's another kind of graph, called a line graph, that you may have seen. These charts have some dots representing, say, monthly sales. The dots are then connected by straight lines. These graphs can give a dramatic impression. You can whip out a chart that shows sales are getting better and better (or one showing you desperately need more and more money).

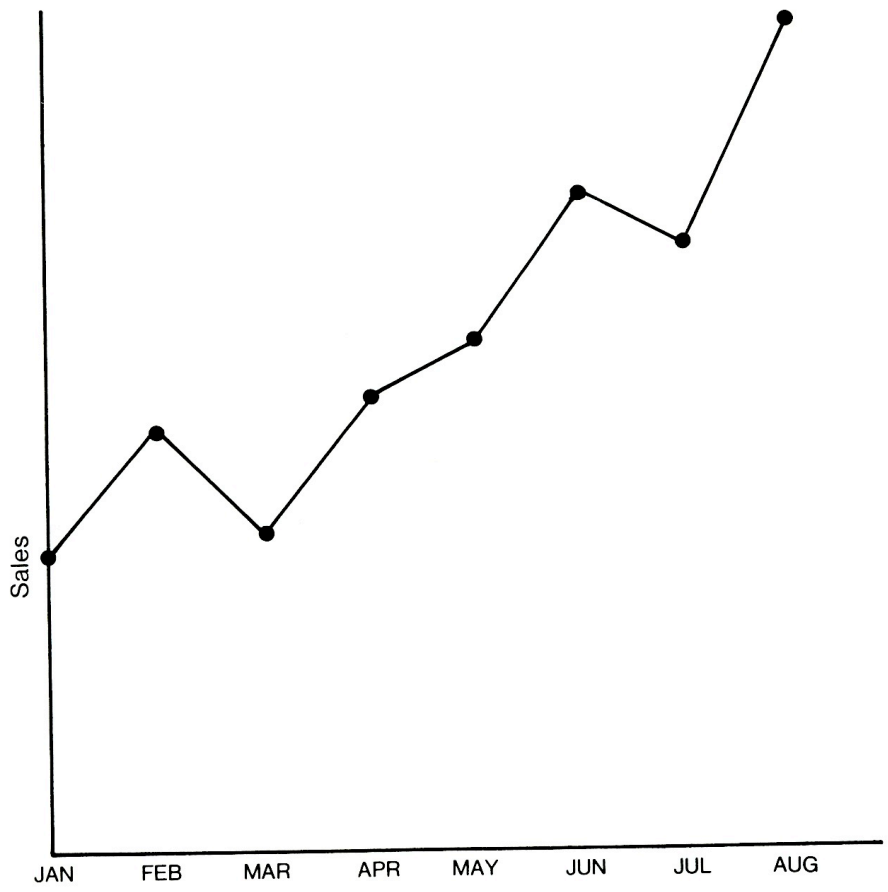


Fig. 14.3

HPLOT TO is the key to drawing this kind of graph. You've already seen some of the preliminary steps like scaling. Once the sales figures have been stored in a list called *sls*( ) and scaled so that the plot fits nicely on the screen, the actual plotting subroutine is surprisingly short.

```
1996 REM ************** line graph ************************
1997 REM  sls( ) is a list of scaled monthly sales
1998 REM
1999 REM ***************************************************
2000 HPLOT 20,158 - sls(1)
2010 FOR mth = 2 TO 12
2020 HPLOT TO 20*mth,158 - sls(mth)
2030 NEXT mth
2040 RETURN
```

Another way of using ADAM's power to connect pixels is to have ADAM draw random shapes. To do this we:

1. Have ADAM randomly decide on the number of sides (at least three and perhaps no more than fifty).

2. Have ADAM randomly decide where to place the sides.

3. Have ADAM HPLOT from one end of a side to another, in order.

Here's the program:

```
10 REM RaNDom shapes
20 HOME
30 PRINT "Press any key to start"
40 GET a$: seed = ASC(a$)
50 r = RND(-seed)
60 sid = INT(47*RND(1)) + 3
70 HGR: HCOLOR = 3
80 GOSUB 500
90 PRINT "Hit s to see a different figure or anything else to
quit"
100 GET st$
110 IF st$ = "s" OR st$ = "S" THEN 70
120 TEXT
130 END
499 REM ********plotting subroutine*******************
500 kl = INT(256*RND(1))
510 rl = INT(156*RND(1))
520 HPLOT kl,rl
530 FOR i9 = 2 TO sid
540 kol = INT(256*RND(1))
550 row = INT(156*RND(1))
560 HPLOT TO kol,row
570 NEXT
580 HPLOT TO kl,rl
590 RETURN
591 REM ************************************************
```

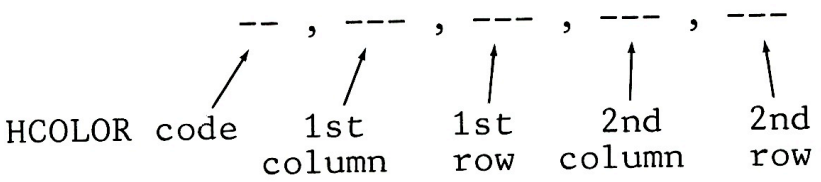
We used the ASC command from chapter 13 to give us the seed for the random number generator. This makes each picture different. Line 60 determines the number of sides—at least three and no more than fifty. The termination routine (lines 90–130) uses the GET command to allow an elegant exit. The most interesting part is the plotting subroutine. We have to place the first point plotted—the one positioned at column *k1* and row

r1—outside the FOR-NEXT loop so ADAM can remember it. Without doing this, there is no way to connect the last side to the first side. We use the HPLOT command (on line 560) to connect the previously plotted point with the new point.

(The INT command, which feeds integers to ADAM for use in the HPLOT command, is actually unnecessary. As with the PLOT command, ADAM truncates any number it encounters before trying to HPLOT.)

Before writing a program to make a line drawing in high-res graphics, sketch the pattern on a grid (graph paper). You will only have to determine the beginning and end points of each line. After this, HPLOT or HPLOT TO will do the rest. This simplifies many high-res picture plots. For example, look at figure 14.4.

As you can see, this rocket ship is made up of seven lines and we've labelled the endpoints. Instead of writing seven HPLOT statements, we'll do it a little differently. The subroutine will read a color code and then read the positions of two points. Then ADAM will HPLOT the line between the two points in the desired color. The information ADAM will use will be placed in a group of DATA statements at the end of the program. Each DATA statement will look like:



We arranged it so that this little subroutine uses the HCOLOR code as an exit. If the code is outside the range of 0 to 15, ADAM will stop plotting. Here's the subroutine.

```
1996 REM *********** HPLOT with DATA ********
1997 REM local variables
1998 REM ko, xl,yl,x2,y2
1999 REM ************************************
2000 READ ko
2010 IF ko > 15 OR ko < 0 THEN 2060
2020 READ xl,yl,x2,y2
2030 HCOLOR = ko
2040 HPLOT xl,yl TO x2,y2
2050 GOTO 2000
2060 RETURN
2061 REM ************************************
```

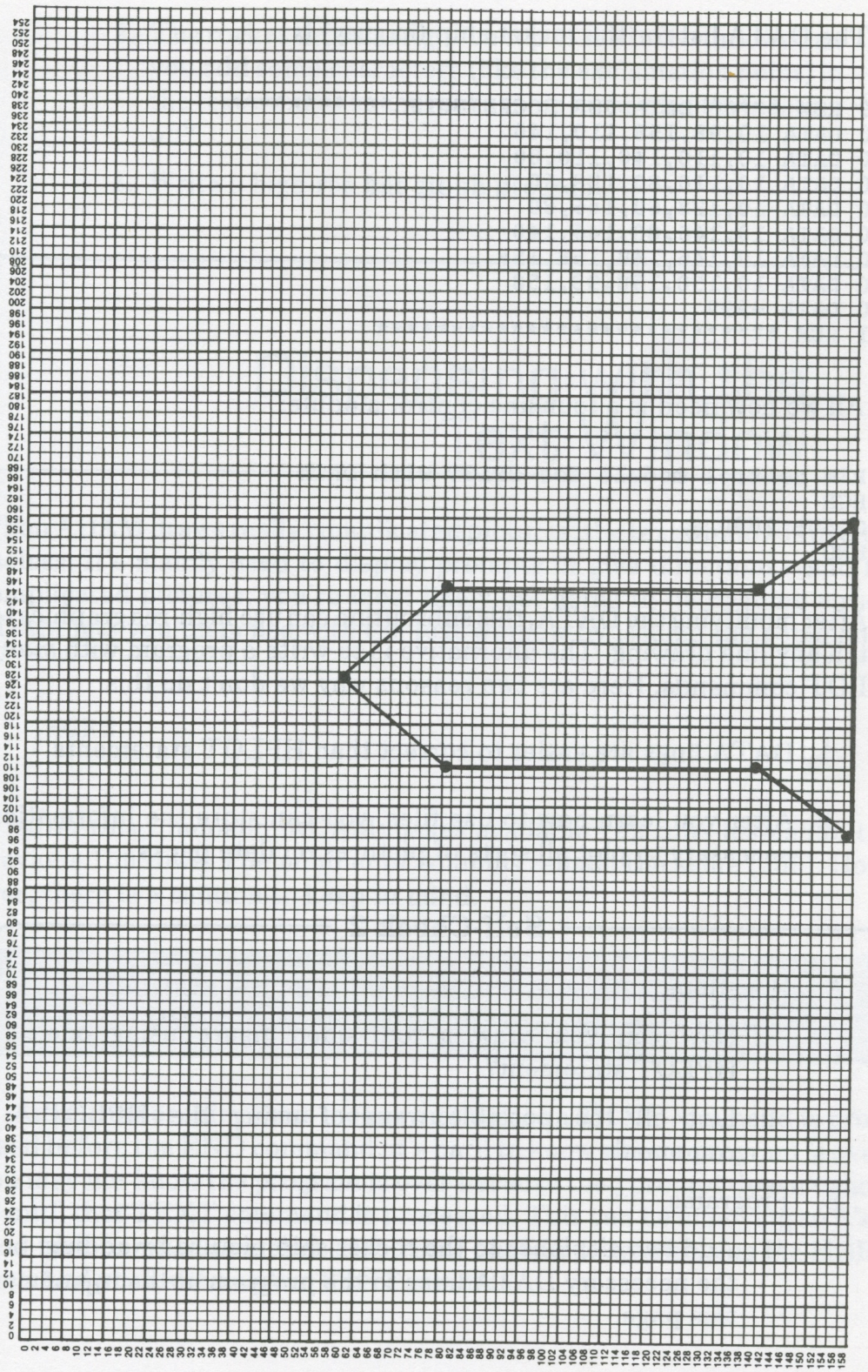


Fig. 14.4

The DATA list for the rocket ship (in grey) is:

```
4999 REM ***** DATA FOR ROCKETSHIP ***********
5000 DATA 10,98,158,110,140
5010 DATA 10,110,140,110,80
5020 DATA 10,110,80,128,60
5030 DATA 10,128,60,146,80
5040 DATA 10,146,80,146,140
5050 DATA 10,146,140,158,158
5060 DATA 10,158,158,98,158
5070 DATA -999
5071 REM ***************************
```

All we now need is a four-line control module:

```
10 REM control module for HPLOT with DATA
20 HGR
30 GOSUB 2000
40 END
```

After modifying the DATA, this subroutine can use high-res graphics to sketch any line drawing. The alternative is to write a program with *many* HPLOT commands. Here are two reasons why we don't do this:

1. DATA lines are easier to correct than HPLOT statements.

2. After you learn how to use files, you can store the information necessary to plot all sorts of pictures on data packs.

Self-Check 2

A. Questions

   1. How can you draw a four-sided figure using one HPLOT statement?

   2. List all the possible ways of using the HPLOT command.

B. Programs

   1. Add two airplanes to the rocket ship picture by adding the necessary DATA lines to the program at the top of this page.

2. The rocket ship sits right on top of the text window. Can you add some characters (like the exclamation point) in the text window so the rocket ship looks like it is blasting off?
3. Write a high-res bar graph program.

Dispensing with the Text Window

When ADAM draws really complicated pictures, the text window is usually useless. It's mainly used to display prompts or give directions—and for complicated pictures, you're most likely not going to be needing either. ADAM lets you dispense with the text window by issuing the command:

```
HGR2
```

After this command, ADAM makes available 192 rows instead of the previous 159 rows. The number of columns remain the same (256). However, there is a caution to keep in mind.

> CAUTION #3: AS LONG AS ADAM IS IN HGR2 MODE, NO MESSAGE OR PROMPTS SHOW UP ON THE SCREEN.

Among other things this means that unless a TEXT command is processed before the END command, it's hard to know when a program is over. You may not know the program is over because the] prompt doesn't show up until that TEXT command is processed. This can be annoying so it's a good idea to *make sure that every route out of a program using the* HGR2 *command passes through a* TEXT *command.*

Suppose we want to write a program to connect random lines, in random colors, to the center of the HGR2 screen. You might be tempted to use the following program.

```
10 REM the joys and dangers of HGR2
20 HOME
30 PRINT "Press any key to start"
40 GET a$: seed = ASC(a$)
50 x = RND(-seed)
60 HGR2
70 HCOLOR = 16*RND(1)
80 xval = 256*RND(1)
90 yval = 192*RND(1)
100 HPLOT 128,96 TO xval,yval
110 GOTO 70
```

If you run this we hope you will find it dramatic and beautiful. However, the program contains an infinite loop. Suppose you press CONTROL/C to stop it. If you look closely, you'll see that the show has stopped *but* you can't be sure that the program is over until you also enter the TEXT command from direct mode. If this isn't enough of a problem you won't be able to see the TEXT command as you enter it. However, since the CONTROL/C combination did stop the program, the] should appear after you enter TEXT.

After stopping an HGR2 *program using a* CONTROL/C, *you must issue the* TEXT *command from direct mode to regain use of the screen.*

Unfortunately, part of the reason this program is so pretty comes from another problem of both HGR and HGR2 graphics.

> CAUTION #4: IF ADAM REPLOTS A DOT TOO OFTEN—WITH MORE THAN TWO COLORS—THE CHIP (INTEGRATED CIRCUIT) THAT ADAM USES FOR ITS HIGH-RESOLUTION GRAPHICS DISPLAY GETS OVERPOWERED AND PLOTS A LARGE RECTANGLE.

For example, if you run the previous program, eventually the display stops consisting of crisp lines radiating out from the center. The lines become blobs.

Figure 14.5 is a grid for ADAM's HGR2 screen. If you intend to work with HGR2 graphics it's worth photocopying.

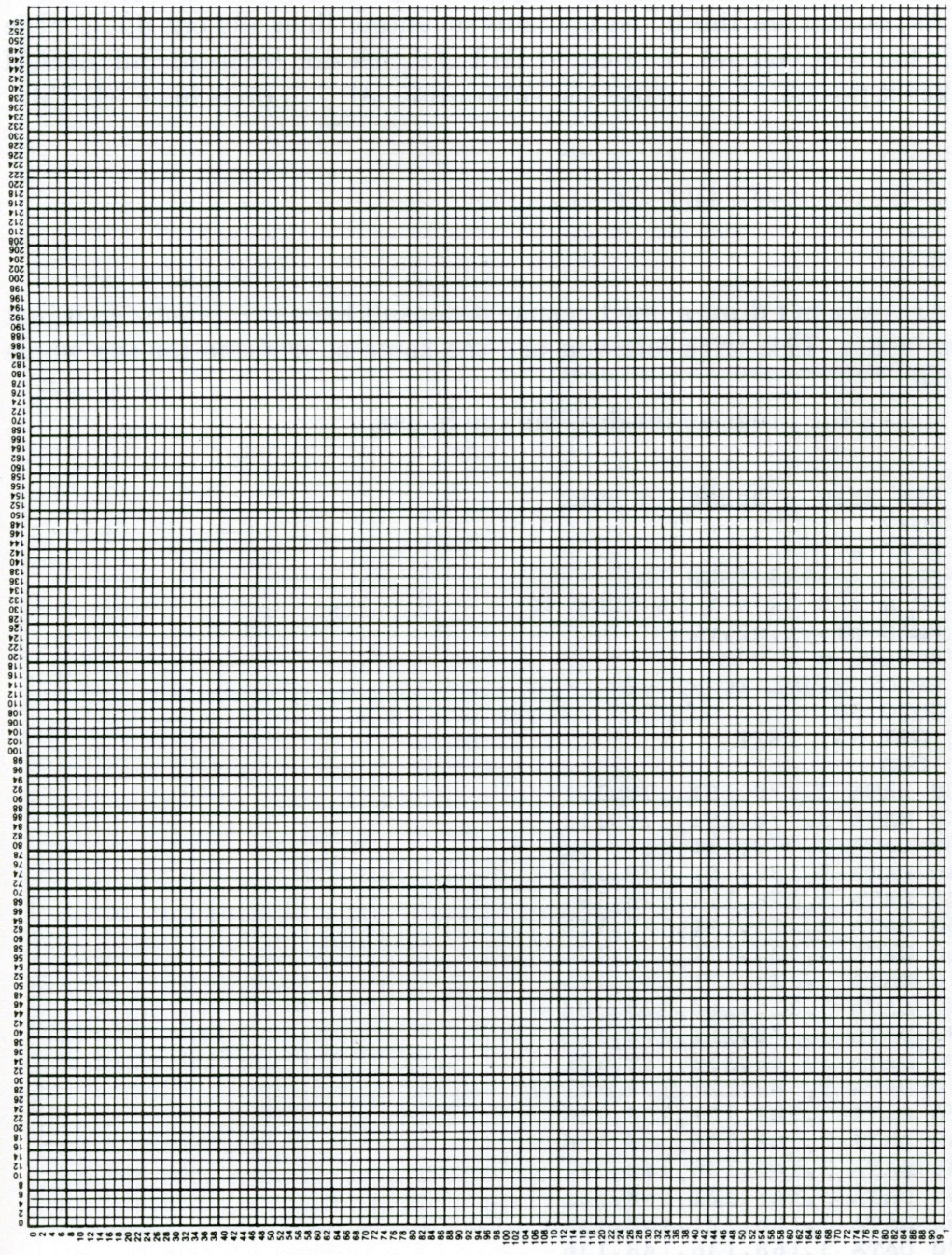


Fig. 14.5 High Resolution (HGR2) Graphics Layout Sheet

Let's use the grid in figure 14.5 and the technique using READ and DATA statements from the last section to draw the picture in figure 14.6.

As you can see, this house contains seventeen lines and one dot (for the door knob of course). You could HPLOT this dot by itself but ADAM makes it possible to still use the subroutine. All you do is have ADAM:

```
HPLOT 125,187 TO 125,187
```

ADAM can't draw a line because it doesn't have two points, so it just plots a dot. This trick lets you use the same subroutine for the whole house.

Here's the program for drawing the house:

```
10 REM draw a house in HGR2
20 HGR2
30 READ a
40 IF a = -999 THEN 100
50 IF a < 0 OR a > 15 THEN 90
60 READ x1,y1,x2,y2
70 HCOLOR = a
80 HPLOT x1,y1 TO x2,y2
90 GOTO 30
100 GET a$: TEXT
110 END
1000 DATA 3,178,191, 78,191
1010 DATA 3, 78,191, 78,111
1020 DATA 3, 78,111,128, 61
1030 DATA 3,128, 61,178,111
1040 DATA 3,178,111,178,191
1050 DATA 3,123,191,123,171
1060 DATA 3,123,171,133,171
1070 DATA 3,133,171,133,191
1080 DATA 3,125,184,125,184
1090 DATA 3, 78,111,178,111
1100 DATA 3, 88,136, 88,121
1110 DATA 3, 88,121,108,121
1120 DATA 3,108,121,108,136
1130 DATA 3,108,136, 88,136
1140 DATA 3,148,136,148,121
1150 DATA 3,148,121,168,121
1160 DATA 3,168,121,168,136
1170 DATA 3,168,136,148,136
1180 DATA -999
```

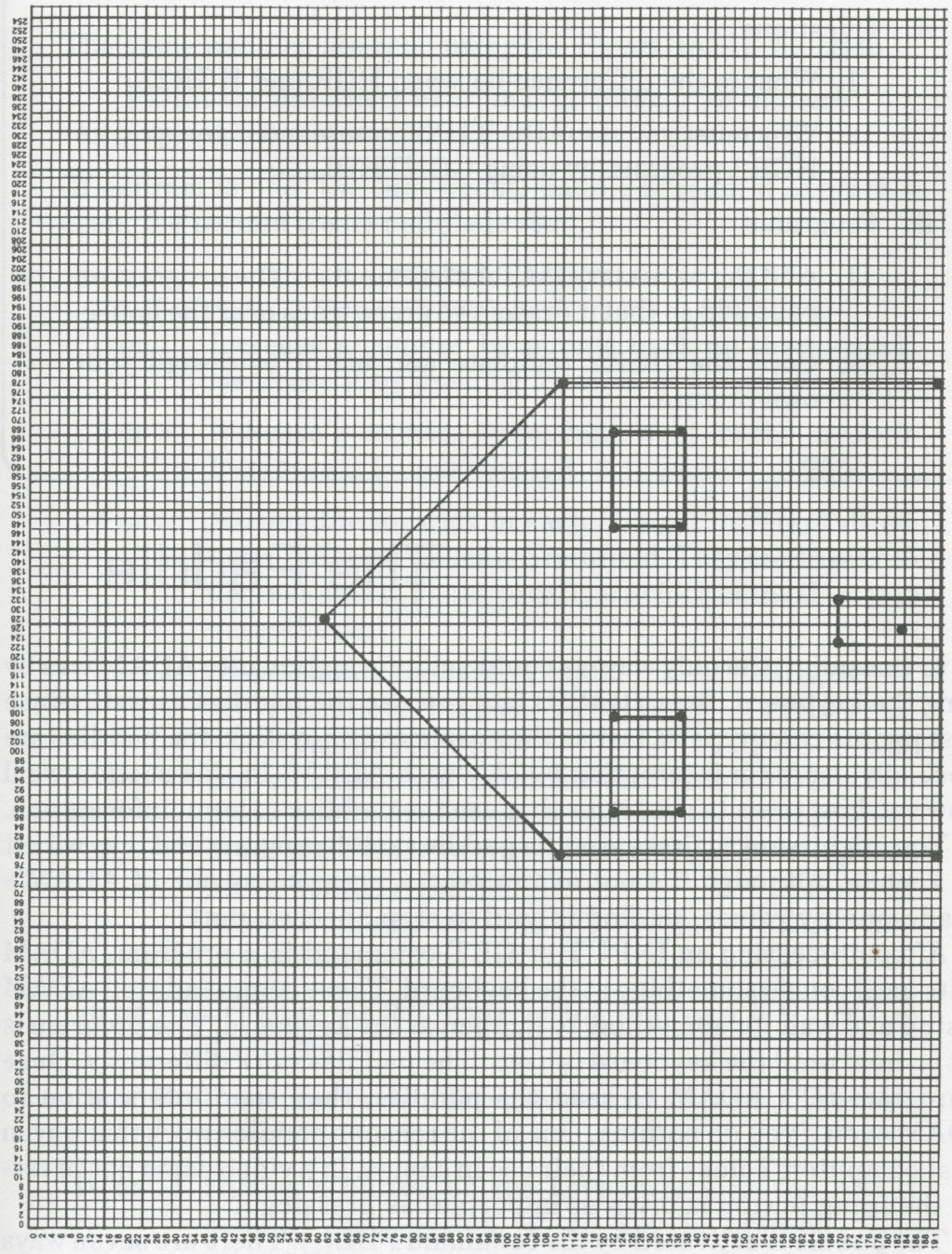


Fig. 14.6 High Resolution (HGR2) Graphics Layout Sheet

Self-Check 3

A. Questions

1. What is the difference between HGR and HGR2?
2. Why must you issue the TEXT command if you stop an HGR2 program by using CONTROL/C?

B. Programs

1. Add the necessary DATA statements to the program on page 368 so that it draws two green trees near the house.
2. Copy the HGR2 template and use it to design your own picture.

The Paddle

Up to this point, the only way we could get information into ADAM was by using the keyboard. SmartBASIC lets you use the paddles as well. Since most people can't type as fast as they can move a joystick, the paddles are especially useful in interacting *quickly* with ADAM. In this section we'll show you how to write programs that use the paddles to input information.

You might want to first enter the paddle sketch program and play with it for a while. Although it's not as involved a program as one of ADAM's arcade-quality games, we find it hard to stop once we start.

After processing an INPUT or GET command in a program, ADAM waits until you do something at the keyboard. These commands tell ADAM to stop and wait for something to happen at the keyboard. The program is put on hold until the keyboard is used. The paddles work differently. Once a program tells ADAM to read the paddles, they send information to ADAM even if you do nothing. One of the reasons paddles work this way in most computers comes from their use in games. In a video game not pressing the firing button is often as important as pressing it.

The paddle is activated by the command PDL( ). There are seven ways of (or functions for) using the paddles, and two paddles. You tell ADAM which of the functions you want to use by putting a number from 0 to 13 inside the parentheses. Table 14.3 summarizes the functions.

Table 14.3 — Paddle Controls

The game paddles are read using the SmartBASIC function:

PDL(exprnm) where exprnm is an integer (or numeric expression) between 0 and 15

(If exprnm is odd, the first (front) paddle is read; if exprnm is even, the second (rear) paddle is read.)

exprnm	action
0 or 1	Increase a counter by pulling back on the joystick. Decrease it by pushing forward. 0 < counter < 255.
2 or 3	Increase a counter by pushing right on the joystick. Decrease it by pushing left. 0 < counter < 255.
4 or 5	Moving the joystick in each of eight different directions, different values are returned. They are spaces 45 degrees apart. The approximate clock equivalent is in parenthesis. Move paddle forward (12:00) PDL( ) = 1 Move paddle forward and right (1:30) PDL( ) =3 Move paddle right (3:00) PDL( ) = 2 Move paddle right and backward (4:30) PDL( ) =6 Move paddle backward (6:00) PDL( ) = 4 Move paddle backward and left (7:30) PDL( ) = 12 Move paddle right (9:00) PDL( ) = 8 Move paddle forward and left (10:30) PDL( ) = 9 Do nothing PDL( ) = 0
6 or 7	Pressing the left firing button returns a 1. Otherwise it returns 0.
8 or 9	Pressing the right firing button returns a 1. Otherwise it returns 0.
10 or 11	Hitting a button on the keypad returns the ASCII code of that button. If no key is pressed, 0 is returned.

(continued)

12 or 13	Hitting a button on the keypad returns the numerical value of that button. * gives 10. # gives 11. If no key is pressed, 15 is returned.
14 or 15	Not yet used.

Table 14.3

For example, looking at the table you see that the forward-back joystick counter for the front paddle is activated by PDL(1). The following short program increases the value of the variable x (up to 255) as long as the joystick is pulled back.

```
10 REM paddle demo #1
20 PRINT "Push the front joystick forward and watch the
numbers decrease"
30 SPEED = 50
40 x = PDL(1): REM assign x the value of the paddle function
50 PRINT x
60 GOTO 30
```

This program must be stopped by CONTROL/C. We set the speed at 50 to slow down ADAM's display. At this speed, line 40 takes longer to process and you get more time to see the effect of the joystick. To set the display back to its normal speed, enter the command:

```
SPEED = 255
```

after you use CONTROL/C.

Here's another example. The PDL(6), PDL(7), PDL(8), and PDL(9) functions tell ADAM whether a firing key has been pressed. You can use them in a program to simultaneously test the reflexes of two different people. The idea is simple. As long as a firing button hasn't been pressed, these PDL functions send the value 0 to ADAM. A program fragment like:

```
5000 IF PDL(6) = 0 THEN ref = ref + 1: GOTO
     5000
5010 PRINT REF
```

has ADAM check that you *haven't yet* pressed the left button on the front paddle. Here's why. Because you haven't pressed the button, the IF clause is false. So ADAM first does the addition and then processes the GOTO 5000 at the end of this line. This GOTO sends ADAM back again to test the IF-THEN. This cycle is so fast that, as soon as you press the button, ADAM will change the value of PDL(6) to 1. Now the IF clause is false and so ADAM moves on to process line 5010. That line prints out your reflex score. The higher the number, the worse you did. Here's the program:

```
10 REM ADAM's reflex tester
20 r1% = 0: r2% = 0
30 HOME
40 VTAB(5): PRINT "Press any key to start"
50 GET a$: seed = ASC(a$)
60 FOR pa = 1 TO 2000*RND(-seed): NEXT pa
70 key = INT(RND(1) + .5)
80 GOSUB 1000
90 GOSUB 2000
100 HOME: VTAB(12): INVERSE
110 IF r1% < r2% THEN PRINT "Player two is the winner"
120 IF r2% > r1% THEN PRINT "Player one is the winner"
130 IF r2% = r1% THEN PRINT "It's a tie!"
140 NORMAL: PRINT: PRINT
150 PRINT "Press a to play again"
160 PRINT "Press anything else to end"
170 GET yn$
180 IF yn$ = "A" OR yn$ "a" THEN 20
190 HOME
200 END
997 REM ******************************************
998 REM  set-up and directions
999 REM ******************************************
1000 IF key = 0 THEN but$ = "left"
1010 IF key = 1 THEN but$ = "right"
1020 HOME: VTAB(12): HTAB(6)
1030 PRINT "Please press and hold the "
1040 INVERSE: VTAB 14: HTAB 14
1050 PRINT but$: NORMAL
1060 PRINT: PRINT "firing button"
1070 RETURN
1997 REM ******************************************
1998 REM         Reflex Tester
1999 REM ******************************************
2000 n6 = 2*key + 6: n7 = 2*key + 7
2010 IF PDL(n6) = 0 THEN r1% = r1% + 1
2020 IF PDL(n7) = 0 THEN r2% = r2% + 1
2030 IF PDL(n6) = 0 AND PDL(n7) = 0 THEN 2010
2040 RETURN
```

There isn't much new in this program. We used integer variables (see chapter 11) because ADAM performs integer addition extremely fast. This makes a tie almost impossible. The firing button that works in the program is chosen randomly. The variable, key, holds the choice of button. This helps prevent "cheating". (Of course you might want to test that no one has both buttons depressed before the game starts, but we'll leave this to you.)

Trick of the Trade You can use the same trick to get a seed for the random number generator. Use a fragment like:

```
1000 PRINT "PRESS THE LEFT FRONT FIRING BUTTON
     TO START"
1010 IF PDL(6) = 0 THEN seed% = seed% -1:
     GOTO 1010
1020 RETURN
```

Since people always respond in slightly different amounts of time, this gives a good way to reseed the random number generator.

We want to use the fourth (or fifth) paddle function to write a paddle sketcher. This program works like the children's toy to draw curves. However, ours is an improved model because we'll let you draw curves in different parts of the screen without having to connect them.

Imagine a bug (this time we mean an insect, not a programming bug!) located at a dot on ADAM's high-resolution grid (either HGR or HGR2). That but can (usually) move in any one of eight directions as seen in figure 14.7.

When the front joystick is moved, PDL(5) sends ADAM one of eight different numbers. The number depends on the direction that the joystick points in. The directions that the bug can move in are exactly the directons that PDL(5) can detect. The main subroutine for a paddle sketcher is simple. Suppose our bug is at a position specified by two variables, *lum* (for the column) and *rw* (for the row). Depending on which way the joystick points, we want to adjust *lum* and *rw*. The line:

```
IF PDL(5) = 1 THEN lum = lum:  rw - 1
```

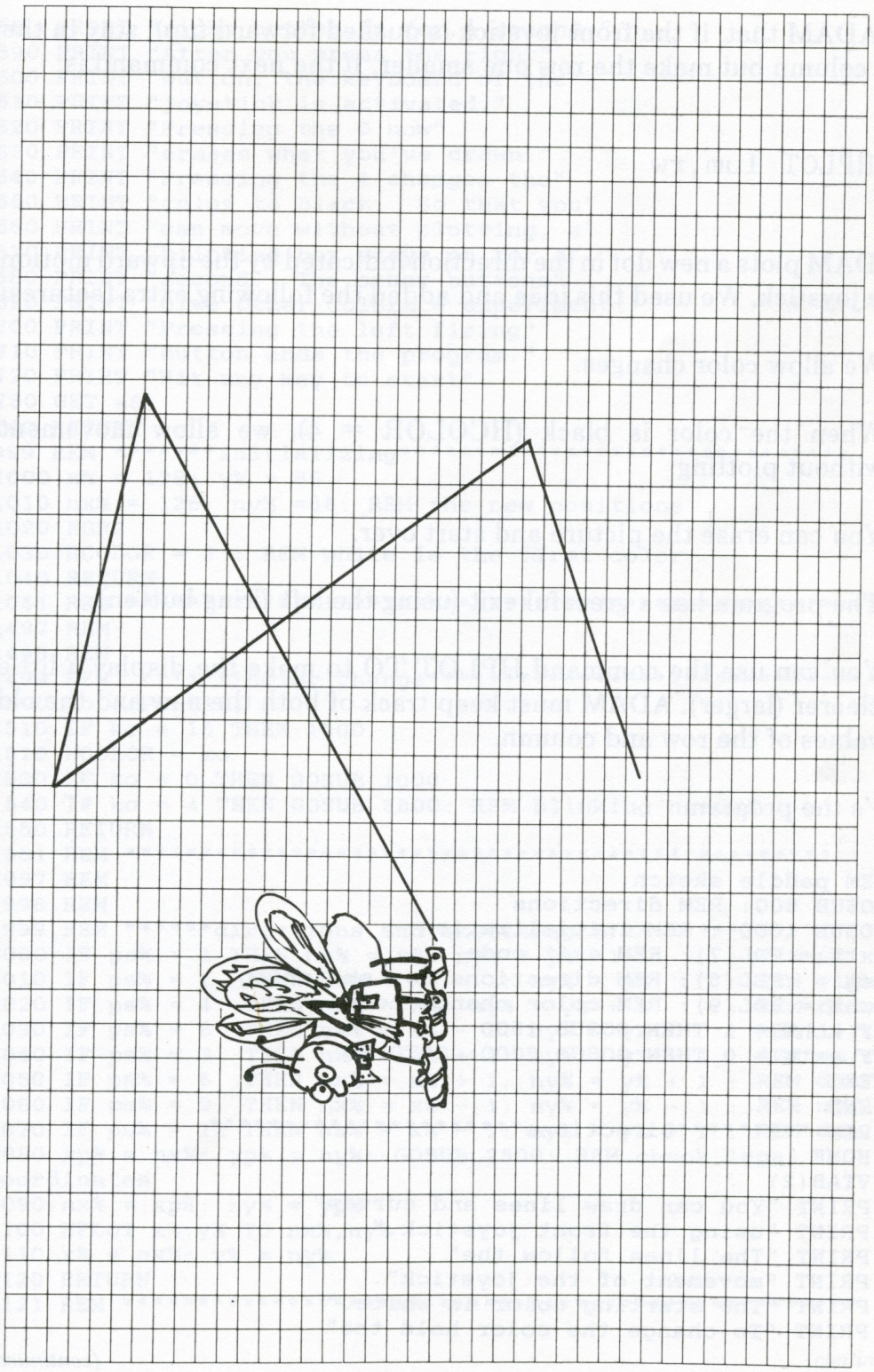


Fig. 14.7

tells ADAM that, if the front joystick is pushed forward (up), stay in the same column but make the row one smaller. If the next command is:

```
HPLOT 1um,rw
```

ADAM plots a new dot in the direction indicated by the upward motion of the joystick. We used this idea and added the following extra features:

1. We allow color changes.
2. When the color is black (HCOLOR = 4), we allow movement without plotting.
3. You can erase the picture and start over.
4. The program has a graceful exit (using the left firing button).
5. You can use the command HPLOT TO to make the display a little clearer (larger). ADAM must keep track of both the new and the old values of the row and column.

Here's the program:

```
10 REM paddle sketch
20 GOSUB 500: REM directions
30 GOSUB 1000 : REM initialization
40 ext% = PDL(7): REM exit code
50 ps% =  PDL(5): REM directions for sketching
60 kch% = PDL(9): REM color change code
70 IF kch% = 1 THEN GOSUB 1500
80 IF ext% = 0 THEN GOSUB 2000: GOTO 40
90 TEXT
100 END
499 REM *********Directions*********************
500 HOME
510 VTAB(2)
520 PRINT "You can draw lines and curves"
530 PRINT "using the front joystick."
540 PRINT "The lines follow the"
550 PRINT "movement of the joystick"
560 PRINT "The starting color is white."
570 PRINT "To change the color hold the"
```

(continued)

```
580 PRINT "right button in for a second."
590 PRINT "After you press the right"
600 PRINT "button, the keyboard of the"
610 PRINT "joystick is activated."
620 PRINT "Pressing the 0 now"
630 PRINT "erases what you've drawn."
640 PRINT "Pressing the 4 changes the"
650 PRINT "color to black.  So that you"
660 PRINT "can move without plotting, a"
670 PRINT "blinking box stays on for a"
680 PRINT "second.  The other buttons"
690 PRINT "give other colors - experiment!"
700 PRINT "Pressing the left firing"
710 PRINT "button ends the program."
720 PRINT "Hit any key to start"
730 GET a$
740 RETURN
999 REM *******initializing*****************************
1000 x% = 128: y% = 86
1010 nx% = 126: ny% =86: REM the new positions
1020 HGR2
1030 HCOLOR = 3 : REM white is the first color
1040 RETURN
1041 REM ************************************************
1497 REM
1498 REM
1499 REM  *******color change*******************************
1500 ko = PDL(13)
1510 IF ko = 15 THEN 1500
1520 HCOLOR = ko
1530 IF ko = 0 THEN GOSUB 1000
1540 IF ko = 4 THEN GOSUB 3500: REM blinking needed
1550 RETURN
1551 REM ************************************************
1997 REM
1998 REM
1999 REM ******directions and sketching********************
2000 IF ps% = 1 THEN nx% = x%: ny% = y% -1       : REM up
2010 IF ps% = 2  THEN nx% = x% + 1: ny% = y%     : REM right
2020 IF ps% = 4  THEN nx% = x%: ny% = y% + 1     : REM down
2030 IF ps% = 8  THEN nx% = x% - 1: ny% = y%     : REM left
2040 IF ps% = 3  THEN nx% = x% + 1: ny% = y% - 1 : REM up/rt
2050 IF ps% = 6  THEN nx% = x% + 1: ny% = y% + 1 : REM dwn/rt
2060 IF ps% = 9  THEN nx% = x% - 1: ny% = y% - 1 : REM up/lft
2070 IF ps% = 12 THEN nx% = x% - 1: ny% = y% + 1 : REM dwn/lft
2080 xp% = nx%: yp% = ny%: GOSUB 2500: REM check legal
coordinates
2090 nx% = xp%: ny% = yp%
2100 HPLOT x%,y% TO nx%,ny%
2110 x% = nx%: y% = ny%
2120 RETURN
2121 REM ************************************************
```

(continued)

```
2498 REM
2499 REM ************legal coordinates?*******************
2500 IF xp% < 0 THEN xp% = 0
2510 IF xp% > 255 THEN xp% = 255
2520 IF yp% < 0  THEN yp% = 0
2530 IF yp% > 191 THEN yp% = 191
2540 RETURN
2541 REM
3499 REM ************blinker*******************************
3500 FOR qw = 1 TO 4
3510 HCOLOR = 3: GOSUB 3600: REM blinking
3520 HCOLOR = 4: GOSUB 3600: REM blinking
3530 NEXT
3540 RETURN
3600 FOR i7 = 0 TO 2
3610 FOR j7 = 0 TO 1
3620 xp% = x% + i7: yp% = y% + j7: GOSUB 2500
3630 HPLOT xp%,yp%
3640 NEXT: NEXT
3650 RETURN
```

None of the parts of this program are very hard. Line 80 in the control module is the key. As long as ADAM doesn't detect that the left button has been pressed, it stays in a loop that first sends it to the plotting module (line 2000–2120). This module—with some time off to check the legality of the coordinates (the subroutine on lines 2500–2540)—does the actual plotting. The RETURN statement on line 2120 sends ADAM to the part of line 80 that says GOTO 40. The process starts all over again.

(Although we think that this program is a lot of fun to use, we want to warn you of one thing. If you change colors a lot you run the risk of overloading ADAM's graphics chip—see Caution #2. Unfortunately, Coleco has not released enough information at this time to know if there is a way to avoid the problem.)

Self-Check 4

A. Questions

1. Why is the paddle better than the keyboard (as used in the paintbrush program on page 176) for entering graphics information?
2. What does PDL(7) test?

B. Programs

1. Write a program that has ADAM tell you what you've just done with the paddle.
2. Modify the paddle sketch so that two people can use it at the same time. (You'd better start them at opposite ends of the screen!)

Summary

In this chapter we introduced you to hi-res graphics and showed you some of the simpler drawings that can be made in hi-res. We showed you how to use the game paddle to send information into ADAM; in particular, we showed you how to draw using the paddle.

We showed you the following commands:

HGR — turns on ADAM's high-resolution graphics screen with a text window.

HGR2 — turns on hi-res graphics with no text window.

HCOLOR — allows you to choose a color to be used by plotting commands that follow.

HPLOT — is the simplest plotting command in hi-res graphics. It has several forms:

- HPLOT *col,rw* — plots a single point in the indicated column (*col*) and row (*rw*).
- HPLOT TO *col,rw* — plots a line from the last plotted point to the point at the indicated *col,rw*.
- HPLOT *c1,r1* TO *c2,r2* — plots a line between positions *c1,r1* and *c2,r2*.

PDL( ) tells ADAM to read the game paddle(s). They are read differently when different numbers are inserted between the parentheses.

BUZZwords

BLEEDING

PIXEL

PICTURE ELEMENT



Getting into (High-Res) Shape

The last chapter built the foundations for high-resolution graphics using SmartBASIC. This chapter gives a sampling of more advanced topics. We'll show you how to manipulate shapes and how to draw curves. Look on page 382 to see an example of what ADAM can draw.

The new commands are:

DRAW	XDRAW
ROT	SCALE

This chapter is less detailed than previous chapters. To completely explain how ADAM handles shapes, or the methods for drawing curves, would fill another moderately long book. We hope a book on this topic will

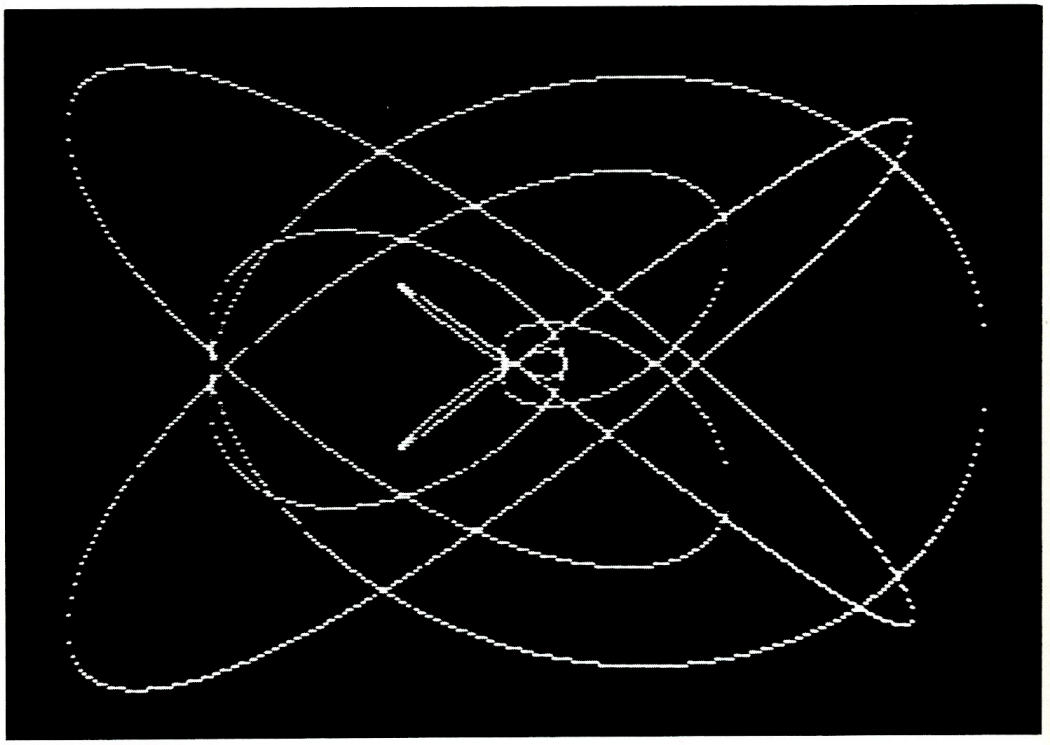


Fig. 15.1

soon be available. We tried to make the programs in this chapter easy to use without your having to know all the details of what makes them tick. One way to use this chapter is to quickly read enough of the text so that you can use the demonstration programs. Only then begin working through the details. However, don't feel that this is the only way. The sections are independent, and can be read in any order that strikes your fancy.

Manipulating Shapes

Chapter 7, on low-resolution graphics, showed you how to animate colored blocks and make it seem as though they moved around the screen. Suppose we try to move figures on the high-res screen. The first idea that comes to mind is to mimic the low-resolution method.

Draw the design in an HCOLOR.

Redraw it in a black (HCOLOR = 0 or 4).

Draw it in a slightly different position.

Unfortunately, again, this won't work! High-resolution pictures are often made up of hundreds (or even thousands) of colored pixels and SmartBASIC, fast as it is, is simply not fast enough. To make this method work requires combining SmartBASIC with ADAM's own internal machine language.

Here's the idea: ADAM lets you store the directions for drawing a shape directly in its memory in a shape table. After the directions for drawing a shape have been stored you can manipulate them in SmartBASIC. When you ask ADAM to draw these shapes you only have to issue one command. You don't need to write hundreds of HPLOT statements. ADAM works much faster after partially bypassing SmartBASIC. The extra speed is usually enough to make animation workable. (Video games are essentially written in ADAM's machine language—that's why they can run so fast.)

The mechanism for getting a shape directly into ADAM's memory is tricky, and has almost nothing to do with BASIC. For this reason we've decided to give you a program in the Afterword that lets you create shapes using the arrow keys. This section only shows you the commands needed to manipulate shapes. To do this we'll work with the one shape built into ADAM. It looks like this:

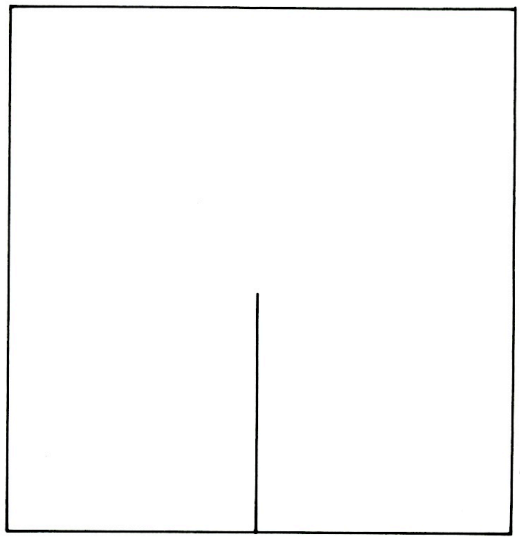


Fig. 15.2 ADAM's default shape

This is usually called the default shape. To make it appear on the screen try the following program:

```
10 REM shape demo 1 - the default shape
20 HGR: HCOLOR = 3
30 SCALE = 1
40 DRAW 1 AT 128,96
50 GET a$: TEXT
60 END
```

If you run this program you should see a small version of figure 15.2 appear on the screen. (To go back to TEXT, press any key.) If you change line 30 to read:

```
30 SCALE = 10
```

and rerun the program, the same design will reappear, only much larger. The SCALE command acts as a magnifier. SCALE = 1 is the smallest size and SCALE = 255 is the largest. Setting the SCALE too large ruins the figure—it becomes too large to fit on the screen and weird things happen. You must set the scale, because the default value of the scale is zero and this will ruin your shape.

Here's what the SCALE command does: if a line in your original shape was ten pixels long and SCALE = 5, it will appear using 50 pixels. The SCALE command expands each pixel by the number you set as the scale.

Each shape stored in ADAM's memory has a number. Since SmartBASIC only comes with one built-in shape, it's number one. You can store up to 255 different shapes at any one time in ADAM's memory. To display any of the shapes stored in memory on the screen, use the command:

```
DRAW _____ AT column,row
```

Replace the _____ by the number of the shape.

Before this command can have any effect ADAM must already be in either HGR or HGR2 mode with both HCOLOR and SCALE set. Here's an example of how to fill the center of the screen with the default shape:

```
10 REM shape demo 2 - multiple copies of the default shape
20 HGR: HCOLOR = 3
30 SCALE = 1
40 FOR lum = 10 TO 245 STEP 10
50 DRAW 1 AT lum,96
60 NEXT
70 GET a$: TEXT
80 END
```

To animate this shape you need a rapid way of erasing it before plotting the next one. The command XDRAW works just like DRAW but it erases the figure. (ADAM redraws the figure in the complimentary color. The complimentary color, when combined with the original color, gives back the background color—black. Unlike APPLESOFT, DRAW must come before XDRAW). Here's a program that uses XDRAW to animate the box:

```
10 REM shape animation 1
20 HGR: HCOLOR = 3
30 SCALE = 1
40 FOR i = 10 TO 250 STEP 2
50 DRAW 1 AT i,96
60 FOR p = 1 TO 200: NEXT
70 XDRAW 1 AT i,96
80 NEXT
90 TEXT
100 END
```

If you run this, the box will appear to run across the screen. This is much faster than ADAM could do the same plotting using the HPLOT command and redrawing it in black. Here's an animation that combines SCALE and XDRAW.

```
10 REM shape animation 2
20 HGR: HCOLOR = 3
30 FOR i = 1 TO 20
40 SCALE = i
50 DRAW 1 AT 128,96
60 FOR p = 1 TO 250: NEXT
70 XDRAW 1 AT 128,96
80 NEXT
90 GET a$: TEXT
100 END
```

There's no reason to stick to white (HCOLOR = 3)—experiment on your own with different colors and scales. To work with a bigger screen, use

the HGR2 command. However, remember that ADAM must be in either HGR or HGR2 mode with an HCOLOR set before the commands SCALE, DRAW, or XDRAW can work.

ADAM gives you other options. Not only can it expand your shapes, you can tell ADAM to rotate them. The command is simple to remember; it's ROT. When:

ROT = 0	Shape appears as you've drawn it.
ROT = 16	Shape is turned 90 degrees clockwise (¼ turn).
ROT = 32	Shape is turned 180 degrees clockwise (½ turn).
ROT = 64	Shape makes a full turn—back to start.

In fact, ADAM will let you use any number between 0 and 255. After 64 the rotations repeat. ROT = 65 is the same as ROT = 1, etc. Each number will rotate the shape 1/64 of a full turn (5.625 degrees if you like degree measurements). If your shape is too small some of these rotations won't do anything. Unlike the SCALE command, ROT doesn't have to be set. The default value is zero, which is a perfectly acceptable rotation.

To write a program that gives a kaleidoscope, all we have to do is:

1. DRAW the figure.
2. Expand, rotate, and change the color.
3. Repeat step 2.

Here's the program:

```
10 REM shape demo 3 - a kaleidoscope
20 HGR2: HCOLOR = 3
30 FOR i = 0 TO 15
40 IF i = 4 OR i = 0 THEN 60: REM black not allowed
50 HCOLOR = i
60 SCALE = 1.6*i + 1
70 ROT = 4*i
80 DRAW 1 AT 128,96
90 FOR p = 1 TO 250: NEXT p
100 XDRAW 1 AT 128,96
110 NEXT
120 GET a$: TEXT
130 END
```

Try manipulating the default shape with various combinations of ROT, SCALE, DRAW, and XDRAW. After you're comfortable with these commands try the shape table generator in the Afterword. By using that program, you won't be restricted to using ADAM's built-in shape.

Self-Check 1

A. Questions

1. What does ROT = 8 do to a shape when it's displayed?
2. Experiment with SCALE. What does SCALE = 0 do to the default figure?

B. Programs

1. Write a program that draws a different colored version of the default shape in each corner of your screen.
2. Write an animation program that makes the default shape seem to move from the left corner towards the right corner. Make it get larger as it moves so that it looks as if it is getting closer to you.

A First Look at Curves

The programming techniques to draw curves in high-resolution graphics need more mathematics than we've used to this point. We don't know a way around this. However, you can draw the pictures in this section using very little mathematics.

Suppose that using HPLOT, ADAM plotted the line connecting two points. Now you want ADAM to move 10 percent of the way down the line. If this line is on the screen it's pretty easy to estimate where this point is. Unfortunately this isn't good enough for ADAM. If you want ADAM to HPLOT starting at this new point then *you* have to give it a formula to use.

Common sense tells us that this point 10 percent of the way down the line should be made up of 90 percent of the first point plotted and only 10 percent of the second.

For example, suppose we ask ADAM to join the two points in column 80, row 160 and column 200, row 100. Where is the point 10 percent of the way along?

90% of the first column is	column 72
90% of the first row is	row 144
10% of the second column is	column 20
10% of the second row is	row 10

So the coordinates of the new point are:

column (72 + 20), row (144 + 10) or column 92, row 154

The following table shows the coordinates of points on the line that are various percentages of the distance down the line.

Point	Percent of the Distance
92,154	10%
110,145	25% (¼ of the way)
140,130	50% (½ of the way)
170,115	75%

Table 15.1

There is a pair of simple formulas to find the points. After you learn them you can program them into ADAM—that's how we made the table. First translate percent into a fraction (ADAM understands fractions but it has no idea what percent means). To translate a percent into a fraction, let *per* equal the percent and divide it by 100.

$$t = per / 100$$

always translates percents into fractions.

For a general formula we start at some point with coordinates *x1*, *y1* (column *x1* and row *y1*) and move in the direction of the point with coordinates *x2*, *y2* (column *x2* and row *y2*). The formulas are:

$$xnew = (1-t)*x1 + t*x2$$

and

$$ynew = (1-t)*y1 + t*y2$$

The new point has coordinates *xnew*, *ynew*.

Using this notion of moving down a line, we can have ADAM draw beautiful pictures. The idea for the following program was shown to us by George Francis of the University of Illinois.

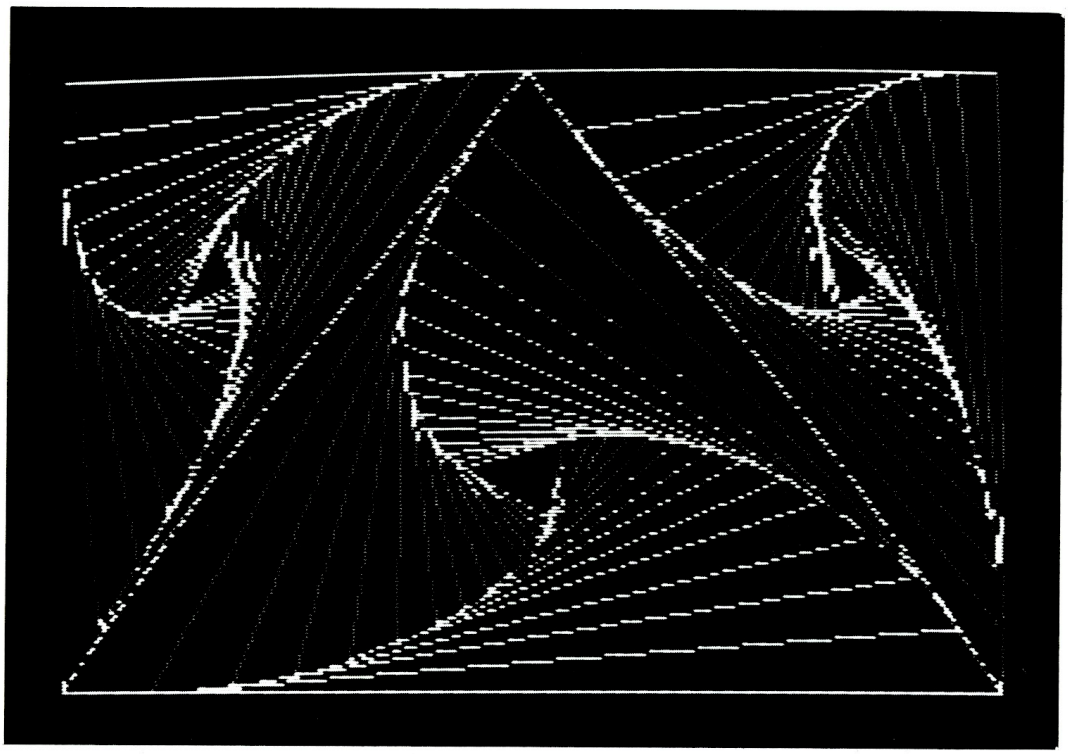


Fig. 15.3

```
10 REM the triangle program
20 HOME: VTAB 10: HTAB 5: PRINT "THE TRIANGLE PROGRAM"
30 VTAB 20: HTAB 5: PRINT "Hit any key to start"
```

(continued)

```
40 GET a$
50 HOME: VTAB 5: PRINT "Choose a tightness"
60 INPUT "(Enter a number between 0 and 1) "; t
70 VTAB 9: PRINT "Enter a color code"
80 INPUT "(A number between 0 and 15) "; co
90 VTAB 12: PRINT "Choose a resolution"
100 INPUT "(Enter a number bigger than 10) "; res
110 VTAB 18: PRINT "Hit any key to start": GET a$
120 HGR2
130 HCOLOR = co
140 RESTORE: GOSUB 1000
150 GET a$: TEXT
160 END
999 REM *********** begin drawing **********************
1000 FOR k = 1 TO 3
1010 FOR i7 = 1 TO 3
1020 READ x(i7),y(i7)
1030 NEXT i7
1040 FOR j = 1 TO res
1050 x(4) = x(1): y(4) = y(1)
1060 FOR i8 = 1 TO 3
1070 HPLOT x(i8),y(i8) TO x(i8+1),y(i8+1)
1080 NEXT i8
1090 FOR i9 = 1 TO 3
1100 x(i9) = (1-t)*x(i9+1) + t*x(i9)
1110 y(i9) = (1-t)*y(i9+1) + t*y(i9)
1120 NEXT i9
1130 NEXT j
1140 NEXT k
1150 RETURN
1151 REM ******************************************************
3000 DATA 125,0,10,175,240,175
3010 DATA 125,0,10,0,10,175
3020 DATA 125,0,240,175,240,0
```

When we run this program, we find that a tightness between .8 and .95 looks best. (*Tightness* measures how close the lines are to each other.) The first time you run it, try color code 3 and resolution 15. After that, we suggest modifying the program so that it will redraw the same picture in different colors. The effects are dramatic. You can do this by placing lines 120 to 140 in a FOR-NEXT loop.

The program does a lot in a few lines. The subroutine has many FOR-NEXT loops. The outer loop (lines 1000 and 1140) repeats all the instructions three times. The best way to understand the program is to forget this loop and look at what happens inside it. The second loop (lines 1010 to 1030) reads the coordinates for the corners of a triangle. The loop on lines 1040–1130 is the key to the program. First lines 1060–1080 plot a triangle. Next the program tells ADAM to move each of the corners a certain percent

of the distance towards the next corner. After this the new (and smaller) triangle is plotted. This keeps repeating and so the triangles shrink in size. The resulting display doesn't look like it comes from lines. It folds into itself. We were amazed the first time we saw the display.

(You might notice that the DATA statements seem to repeat. It was easier to reenter the coordinates than have the program compute the vertices of each of the triangles. You actually need only the coordinates of four points.)

You can use the idea of moving down a line to design your own pictures. The way we learned this method was to think of the following analogy (and look at a picture like Figure 15.4). Suppose you have a tortoise chasing a hare. Now imagine the hare moves away. Draw a line connecting the tortoise to the hare's new position. Now imagine the tortoise moves a little towards the hare on this line. Continue this chase. If the hare always moves in the same direction you get a picture of a curve called a parabola.

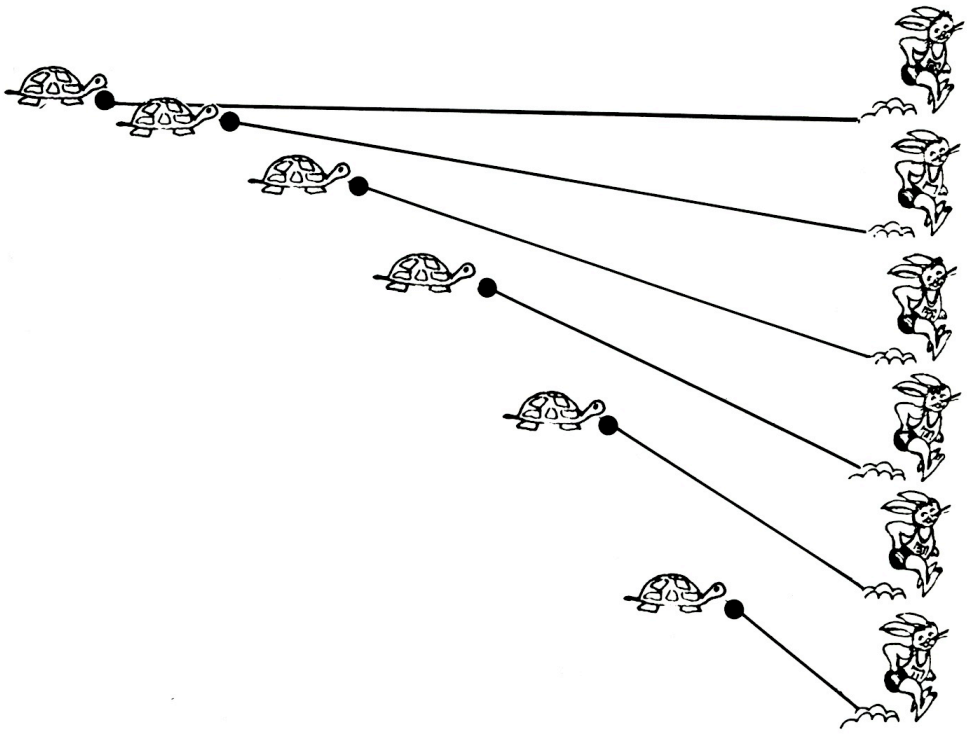


Fig. 15.4

By having more and more animals chasing each other, at different speeds in different directions, the pictures can be more complicated and beautiful. Here's a program that draws the parabola.

```
10 REM the tortoise and the hare
20 HGR2: HCOLOR = 3
30 xt = 10: yt = 10: REM  tortoise's original position
40 xh = 200: yh = 10: REM hare's original position
50 p1 = xt: p2 = yt: GOSUB 500
60 HPLOT xt,yt
70 HPLOT xh,yh
80 GOSUB 1000
90 GOTO 50
100 GET a$
110 TEXT
120 END
499 REM **********error test*************
500 IF p1 < 0 OR p1 > 255 THEN 100
510 IF p2 < 0 OR p2 > 191 THEN 100
520 RETURN
997 REM *********************************
998 REM       New Position
999 REM *********************************
1000 xh = 200: yh = yh + 3: REM hare moves away
1010 xt = .95*xt +.05*xh: yt = .95*yt + .05*yh
1020 RETURN
1021 REM ********************************************
```

The subroutine on lines 1000–1020 is crucial. Line 1000 gives the new position of the hare. Notice that the hare always moves three pixels straight down. Line 1010 uses the formulas at the beginning of this section to calculate the position of the tortoise after it moves 5 percent of the way on the line connecting the tortoise to the hare. You can vary the speed and direction of the animals by changing lines 1000 and 1010.

There's another possibility; instead of using HPLOT *xt*, *yt* on line 60, change it to:

```
HPLOT xt,yt TO xh,yh
```

This gives a solid parabola by connecting the hare's position to the tortoise's. If you also randomly change the colors, the resulting picture is quite striking. (It resembles a multi-colored arch.)

Here are some hints on changing this program to draw more complicated pictures. First imagine more animals chasing each other. This

might make the programming too complicated *except* that each animal moves along some path connecting it to the animal directly in front. This means you never have to calculate the positions of more than two animals at one time. It's worth experimenting with this on your own. For example, have the animal(s) being chased move in directions other than straight down (for example, randomly or in circles).

Self-Check 2

A. Question

1. On the line connecting the points with coordinates 100,10 to 0,150, what are the points 10 percent of the way down? Fifty percent? Seventy-five percent?

B. Programs

1. Suppose a hare is chasing a fox who moves down one row at a time. The hare now chases the fox by moving 20 percent of the way down the line connecting the hare to the fox. The tortoise is still slowly chasing the hare as in the last program. How would you change the program so that it draws the new position of the tortoise?
2. Redo the same program with the fox moving randomly.
3. Try different combinations of animals moving at different speeds in different directions.

Sketching Circles and Other Curves

In this section we'll show you how to graph almost anything provided you have the right formula. Although finding a formula to graph a particular curve may involve difficult mathematics, once you know the formula, telling ADAM to HPLOT it isn't hard. For example, we'll give you a program to graph many of the functions you're likely to encounter in school.

(These are the rational functions.) All you'll have to do is enter it and follow directions.

If the terms or formulas we use in this section seem new or too difficult we suggest skipping the discussions. Try the programs, though, many of them give pretty or useful results. Finally, to follow our discussions you should make sure you understand the notion of relative coordinates (explained in chapter 11).

Suppose you have decided that each pixel represents one unit. Then the HGR2 screen will give an X axis that allows x values ranging from −127 to +127. The y values range from −95 to +95. Most of the time you'll want to change this scale. For example, a scale of 10 pixels for each unit will allow numbers between −12.7 and +12.7 on the X axis.

In school or work, you may need the graph of some complicated functions like:

$$y = x^3 - x^2 + 7$$

or:

$$y = \frac{x + 1}{x^2 - 2x - 3}$$

Using ADAM's HGR (or HGR2) screen you can see pictures of these functions almost immediately. The first trick is to get the function into ADAM. (We showed you how to do it in chapter 12.) After that the outline of the program is:

1. Decide how many pixels represent a unit on each axis.
2. Change each column index by this scale to find the x coordinate.
3. Calculate the corresponding y value using a function (or subroutine call).
4. Change the y value, using the scale factor, into a row index.
5. HPLOT the point using the above column and row values.

Let's work through an example. Suppose the function is something like:

$$y = x^2 + 15x + 23$$

and the scale is 10 pixels to the unit. We start seeing what happens in column 0. Using one pixel for each unit means that column 0 corresponds to −127 on the x axis. Now we have to divide by the scale. So:

column 0 $\qquad xval = -127 / 10 = -12.7$

Now calculate the function:

$$(-12.7)^2 - 15(-12.7) + 23 = -5.21$$

Now change this by the scale to find (approximately) what row this is on. The result, −5.21, is now (about) 52 rows down from the center row. This means that we should tell ADAM to:

```
HPLOT 0,96 + 52.
```

Obviously you wouldn't want to do this by hand for all 255 columns. If ADAM does these calculations we'll need to check that the row to be plotted is actually on the screen *before* ADAM tries to HPLOT the point (otherwise the program will bomb). The following program adds two extra features. First, after ADAM finishes the program, it returns to TEXT mode after you hit any key. ADAM then asks you if you want to replot the same function using a different scale. This lets you put parts of the graph of the function under a "microscope". The larger the scale, the more accurate the graph will be near the origin (the point with coordinates 0,0). The second feature is to allow functions with denominators (rational functions).

```
10 REM rational function graphs
20 HOME
30 GOSUB 500: REM directions
```

(continued)

```
40 INPUT "Enter your scale "; sc
50 HGR2
60 HCOLOR = 14
70 HPLOT 0,96 TO 255, 96
80 HPLOT 128,0 TO 128,191
90 GOSUB 1000
100 GET a$: TEXT: VTAB 5
110 PRINT "Do you want to change the     scale?"
120 PRINT "(Press y or n) "
130 GET yn$
140 IF yn$ = "y" OR yn$ = "Y" THEN 40
150 END
497 REM ************************************************
498 REM     Numerator and Denominator
499 REM ************************************************
500 INPUT "What is the degree of the     numerator? ";num
510 DIM t(num)
520 FOR i = num TO 0 STEP -1
530 PRINT "What is the coefficient of    the term of degree
"; i ;" (Start from the highest power)"
540 INPUT t(i)
550 NEXT i
560 INPUT "Is there a denominator (Enter y or n) ";yn$
570 key = 0
580 IF yn$ = "N" OR yn$ = "n" THEN key = 1: GOTO 650
590 INPUT "What is the degree of the denominator?";den
600 DIM b(den)
610 FOR i = den TO 0 STEP -1
620 PRINT "What is the coefficient of    the term of degree
"; i ; "(Start from the highest power)"
630 INPUT b(i)
640 NEXT i
650 RETURN
997 REM ************************************************
998 REM       plot routine
999 REM ************************************************
1000 FOR p = 0 TO 255
1010 xval = (p-128)/sc
1020 GOSUB 2000: REM denominator
1030 GOSUB 3000: REM numerator
1040 IF bot = 0 THEN 1090
1050 yval = 96 -(sc*top/bot)
1060 yval = INT(yval + .5)
1070 IF yval < 0 OR yval > 191 THEN 1090
1080 HPLOT p,yval
1090 NEXT p
1100 RETURN
1997 REM ************************************************
1998 REM       calculate denominator
1999 REM ************************************************
2000 IF key = 1 THEN bot = 1: GOTO 2050
2010 bot = 0
```

(continued)

```
2020 FOR i7 = 0 TO den
2030 bot = b(i7)*xval^i7 + bot
2040 NEXT
2050 RETURN
2997 REM ************************************************
2998 REM       calculate numerator
2999 REM ************************************************
3000 top = 0
3010 FOR i7 = 0 TO num
3020 top = t(i7)*xval^i7 + top
3030 NEXT
3040 RETURN
```

X-Y coordinates describe points by telling you how much across and up or down they are from the origin. You may have seen one other method for describing points. With polar coordinates you describe points by using two different numbers. The first measures the distance from the point to the origin. The second is the angle between the X axis and the line connecting the point to the origin.

Polar coordinates describe points in the plane by telling you:

1. What size circle, centered at the origin, the point is on.

2. Where, on that circle, the point is located.

To go back from polar coordinates to X-Y coordinates, use the formulas:

$$x = r * \cos \theta$$

$$y = r * \sin \theta$$

where θ is the angle.

All this wouldn't be of much interest if many beautiful curves (for example, ones resembling hearts or roses) weren't best described and most easily drawn using polar coordinates. Even circles are most easily described in polar coordinates. We want to explain, by examples, how to write a subroutine to graph any equation given in polar coordinates. If you ever are given such an equation in school or work, this will show you what to do.

Suppose you want to graph a four-leaf clover of reasonable size. Its equation turns out to be:

$$r = 60*\cos(2*\theta)$$

where θ is an angle that makes one complete revolution.

Before we write the program, recall (from chapter 9) that ADAM uses radian measure and not degrees. This means the angles must run between 0 and 2π radians (and not 0 and 360 degrees). Here is an outline for the program:

1. Set up a function to compute r for many different angles (for example, every hundredth of a radian).

2. Using the formulas:

   $x = r * \cos \theta$

   $y = r * \sin \theta$

   find the ordinary x and y values.

3. HPLOT the relative coordinates $128 + x, 96 - y$.

4. Do this for all the angles.

Here's a program for the four-leaf clover:

```
10 REM a four leaf clover
20 HGR2: HCOLOR = 3
30 pi = 4*ATN(1): REM define pi = 3.14....
40 DEF FN r(ang) = 60*COS(2*ang)
50 FOR ang = 0 TO 2*pi STEP .01
60 xval = FN r(ang)*cos(ang)
70 yval = FN r(ang)*sin(ang)
80 HPLOT xval + 128,96 - yval
90 NEXT
100 GET a$: TEXT
110 END
```

Polar coordinates are the best way to graph circles or parts of circles. For example a circle of radius 60 is described by the polar equation:

$$r = 60$$

The subroutine given below will let you:

1. Put the center of the circle anywhere on the screen.

2. Fill in the circle on demand.

3. Change the radius on demand.

We'll explain it in the trick of the trade that follows. Here's the subroutine:

```
990 REM ************************************************
991 REM global variables: rd = radius: xc = column of
992 REM center: yc = row of center: fi = the `fill in'
993 REM key
994 REM local variables: xval, yval, pi
995 REM
996 REM This routine assumes the HCOLOR is already set and
997 REM ADAM is in HGR2 mode
998 REM
999 REM *************CIRCLE SUBROUTINE******************
1000 GOSUB 1500: REM check legality
1010 pi = 4*ATN(1)
1020 FOR th = 0 TO pi step .01
1030 xval = rd*COS(th)
1040 yval = rd*SIN(th)
1050 IF fi = 0 THEN HPLOT xc + xval,yc - yval:
HPLOT xc + xval,yc + yval
1060 IF fi = 1 THEN HPLOT xc + xval,yc - yval TO
xc + xval,yc + yval
1070 NEXT
1080 RETURN
1241 REM ************************************************
1498 REM
1499 REM **************ERROR CHECK*********************
1500 IF xc - rd < 0 OR xc + rd > 255 OR yc - rd < 0
OR yc + rd > 191 THEN POP
1510 RETURN
1521 REM ************************************************
```

Tricks of the Trade In terms of programming, the first (sub) subroutine call is the most interesting. We'll start there. The point worth noticing is why we used the POP command when the radius was impossible. As we mentioned in chapter 12, after a POP command ADAM forgets the first place it was to return to. This means that if the radius is illegal the RETURN command on line 1510 would not send ADAM back to line 1010 but rather

to whatever line called the circle subroutine in the first place. Using a POP this way is quite common—it prevents illegal subroutine calls.

Since we wanted to have the subroutine let ADAM plot circles anywhere on the screen, we used coordinates relative to the new center. Instead of using:

```
HPLOT 128 + xval,96 - yval
```

we used:

```
HPLOT xc + xval,yc - yval
```

where *xc* is the column of the new center and *yc* is its row. To let the subroutine plot circles or disks, we sent the subroutine a key that tells it whether to fill in the circle.

This subroutine can be used in many ways. For example, to draw a single circle, use:

```
10 REM circle
20 HOME
30 PRINT "Enter the size of the circle"
40 INPUT rd
50 INPUT "ENTER the row and column "; xc,yc
60 PRINT "If no circle shows up then ADAM can't
   draw that circle on its screen - it's too big or too small"
70 PRINT "After the circle is finished press any key to
   erase it"
80 PRINT "Press any key to start"
90 GET a$
100 HGR: HCOLOR = 3
110 GOSUB 1000
120 GET a$: TEXT
130 END
```

You can draw some spectacular pictures by calling the subroutine in various kinds of FOR-NEXT loops. For example, you can draw many circles on the same screen. Properly spaced, they give a nice three-dimensional effect. The control module for this must be a little different.

The radii remain the same but the centers change each time. In this control module we kept the circles in the same row (same *yc*) but moved the column (*xc*) on each pass in the loop.

```
10 REM offset circles
20 HGR2: HCOLOR = 3
30 xc = 50: yc = 96: rd = 50
40 FOR xc = 50 TO 200 STEP 8
50 GOSUB 1000
60 NEXT
70 GET a$: TEXT
80 END
```

Experiment with different ways of calling the subroutine—the pictures can be striking.

Tricks of the Trade We discovered ADAM's inability to plot many colors over the same dot using the circle subroutine. For a dramatic demonstration of ADAM's relative weakness, use the following short control module that tries to plot one colored circle inside another:

```
10 REM a weakness of ADAM's graphics
20 HGR2
30 xc = 128: yc = 96: fi = 1
40 rd = 60: HCOLOR = 2
50 GOSUB 1000
60 rd = 40: HCOLOR = 14
70 GOSUB 1000
80 GET a$: TEXT
80 END
```

The inner circle will show up as a bunch of rectangles *trying* to be a circle.

Another point worth thinking about is that there are more columns than rows; because of this, the circles may appear slightly egg-shaped. If you're not satisfied, it's easy to rescale the variable *yval* in the original circle subroutine.

Self-Check 3

Programs

1. Use the circle subroutine to draw four small circles in each of the corners of the screen.

2. Write a control module that starts out with a tiny circle in the left-hand corner and expands it.

The following demonstration program shows some of the pictures that result from random polar equations. If you like the results, the trick of the trade that follows the program listing explains what goes into it.

```
10 REM  demonstration program for polar coordinates
20 pi = 4*ATN(1): a = 1: b = 1: REM initialize
30 HOME: VTAB (12)
40 PRINT "This demonstrates my drawing"
50 PRINT "abilities.  Press any key to"
60 PRINT "start": PRINT
70 GET a$: seed = ASC(a$)
80 x = RND(-seed)
90 GOSUB 500
100 GOSUB 1000
110 HGR2: HCOLOR = 3
120 GOSUB 1500
130 GET a$
140 TEXT
150 END
499 REM ********** directions *************************
500 HOME: VTAB (7)
510 PRINT "Do you want it to be really"
520 PRINT "complicated?  Press y or n."
530 GET yn$
540 IF yn$ = "y" OR yn$ = "Y" THEN a = INT(12*RND(1)+1):
b = INT(12*RND(1)+1): GOTO 580
550 IF yn$ = "n" OR yn$ = "N" THEN 580
560 PRINT "You pressed something strange"
570 GOSUB 700: GOTO 500
580 PRINT: PRINT: PRINT "How much time do you want me"
590 PRINT "to spend drawing?"
600 PRINT "Press a number from 0 (fastest)";
610 PRINT "to 9 (slowest)."
620 PRINT: PRINT "After the drawing stops press"
630 PRINT "any key to go back to TEXT"
640 GET tep$: tep = VAL(tep$)
650 tep = .046-(tep*5E-03)
660 RETURN
700 PRINT "Press s to stop"
710 GET sn$
720 IF sn$ = "s" OR sn$ = "S" THEN  140
730 RETURN
999 REM ****** define functions ***********************
1000 al = INT(7*RND(1)) + 1
```

(continued)

```
1010 bl = INT(6*RND(1)) + 1
1020 cl = INT(6*RND(1)) + 1
1030 dl = INT(3*RND(1)) + 1
1040 el = INT(4*RND(1)) + 1
1050 fl = INT(3*RND(1)) + 1
1060 IF bl = 1 THEN DEF FN r(th) = 1 + SIN(al*th): mc =  2
1070 IF bl = 2 THEN DEF FN r(th) = dl + SIN(al*th): mc = 1+dl
1080 IF bl = 3 THEN DEF FN r(th) = SIN(cl*th): mc = 1
1090 IF bl = 4 THEN DEF FN r(th) = 1 + dl*COS(el*th):
mc = 1+dl
1100 IF bl = 5 THEN DEF FN r(th) = el*SIN(th)+fl*COS(th):
mc = el+fl
1110 IF bl = 6 THEN DEF FN r(th) = el*SIN(th)/(2-COS(th)):
mc = el
1120 RETURN
1499 REM ****** calculate functions ********************
1500 tpi = 2*pi
1510 FOR th = 0 TO tpi STEP tep
1520 xval = 126/mc*FN r(th) * COS(a*th)
1530 yval = 96/mc*FN r(th) * SIN(b*th)
1540 xval = INT(xval + .5)
1550 yval = INT(yval + .5)
1560 HPLOT 127 + xval, 96 - yval
1570 NEXT
1580 RETURN
```

Trick of the Trade If you hit the **n** key and ask for a relatively uncomplicated picture, this program randomly graphs one of the standard polar graphs. (Determined on lines 1000–1060). If you ask for a complicated picture, the program uses a modification of normal polar coordinates. Instead of:

$$x = r*\cos\theta$$

and:

$$y = r*\sin\theta$$

we use:

$$x = r*\cos(a*\theta)$$

and:

$$y = r*\sin(b*\theta)$$

This makes the pictures much more complicated, and so, more fascinating.

Finally, if you know the addition formulas for sine and cosine you can use them to speed up all of these programs. We'll leave the details to you.

Summary

In this chapter we showed you a sampling of advanced topics in high-resolution graphics. The new commands were:

DRAW_____AT column, row	to draw a pre-defined shape on the screen at the location indicated.
XDRAW_____AT column, row	to draw the shape in its complimentary color at the location indicated.
SCALE	to change the scale used in a DRAW (or XDRAW) command.
ROT	to rotate a shape before displaying it.

BUZZwords

DEFAULT SHAPE SHAPE TABLE



The Data Pack— Your Filing Cabinet

In this chapter we will start using ADAM's data packs as a master filing system. You'll learn the commands that control filing operations and some of their limitations.

For most programmer's learning to handle data files isn't easy. To help you over these difficulties, we've put the basic methods in the main text of this chapter and left the fancy features like techniques to debug files to various tricks of the trade that are set off in boxes. We suggest working through the main body of the chapter before tackling these tricks of the trade. The new commands are:

PRINT CHR$(4)	READ	WRITE
OPEN	RESUME	RENAME

ONERR GOTO	ERRNUM(1)	RECOVER
APPEND	CLOSE	

The first section gives some basic information about files. The next section shows how to read and write simple information into data files. The following two sections show you some long programs that use files, and the last two sections act as a bridge to more advanced techniques. These require the following new commands:

INIT	LOCK	MON
UNLOCK	NOMON	

One caution before starting; up to this point the best way to learn about ADAM has been to modify our programs or to experiment on your own. With files this is a little dangerous. If you write a program using files that seems to crash your machine, look through this chapter (especially at the tricks of the trade) before assuming your ADAM is broken. There's a good chance that you only broke one of the rules for handling files. It's because there are so many rules that we've decided to show you two ways to handle files. As we said, the main body of this chapter deals with the basic method, which covers most situations. The tricks of the trade go into some of the subtleties.

Introduction and Review

In chapter 2, we compared a computer to an office. In that office any information to be kept for more than one day needed a special storage area—the filing cabinet. ADAM has the same problem. Information, whether programs or data, that is needed for more than one session at the keyboard must be stored somewhere. When ADAM is turned off, everything in its memory disappears.

By now you have probably written programs that took hours to get running correctly. You usually want to use those programs more than once. If a program has been saved, it can be run as many times as you want and

whenever you want. Suppose a program uses a large amount of data. ADAM gives you many options for entering this data: assignment or DATA statements built into the program, or repeated use of INPUT statements. The last choice requires reentering the data each time you run the program. The more data needed, the more impractical each of these methods becomes. Fortunately, the data packs can store this information. After information is stored there, you can change it, use it, and, when you're done, throw it away. Data packs can also hold the information that comes out of your programs. This lets you save results of complicated or time-consuming programs.

We'll quickly summarize the steps involved in saving a program. If SmartBASIC isn't loaded into ADAM, insert the SmartBASIC tape and hit the RESET button. Take out that data pack and insert any other data pack that has programs stored on it. In immediate (direct) mode, type CATALOG and hit RETURN. ADAM then displays the contents of the data pack on the screen. Some volume name or number heads the list—this means absolutely nothing to ADAM but you can use it to identify different data packs (see the INIT command in the last section of this chapter). Further down the screen you see a list of the names of the files stored on the data pack. Files are like file folders—each one contains a separate program or a separate collection of pieces of information. There is a letter before each file name. The letters you should see are capital and lowercase A's and H's. The A is the code for ASCII files. These are files containing programs or any other information written in English language characters using SmartBASIC. The H's usually indicate files prepared in (or stored by) the word processor and binary files. The lowercase letters indicate backup files (see below). The number next to the name measures how much space that file takes up.

The space on a data pack is measured in blocks. Each block can contain about one double-spaced page (1K or 1024 bytes—see chapter 2) of information. Even short files use up a block of space. The same thing happens when you write one telephone number on an index card. A lot of space may be wasted but it makes organizing the numbers easier. Each data pack can store 253 blocks of information. There is room to store a tremendous amount of information. For example, the longest program we wrote, while exploring ADAM, was 16 blocks long and that was a monster. Since each data pack has 253 blocks, you might think that a data pack could hold 253 short programs or data files. Unfortunately, this is not so. You see the reason for this when you use the SmartWriter word processor to get a file.

After you press the GET key, ADAM displays:

```
one momment - getting directory
```

and soon shows you a table with a list of names. This directory holds the names of all data and program files stored on the data pack. Since the directory only has room for thirty-five names, each data pack holds only thirty-five files. It's possible, though unlikely, to store thirty-four short files—each one block long—and one monstrous file 229 blocks (229 pages) long. When you enter the command CATALOG in SmartBASIC, you are asking ADAM to access and display the directory. (*Access* is the buzzword meaning get hold of.)

Anyone who programs computers should know that programs and data stored on disks and data packs can, unfortunately, be destroyed. Failure of the drive or accidental erasure of a disk or tape can create incredible grief. Even a speck of dust in the wrong place on a data pack can make it unusable. Repeated use of a data pack will wear it out—whether or not you take perfect care of it. (We don't want to scare you *too* much. A data pack should last for hundreds, even thousands, of plays.) You can't avoid the problem but you can avoid the grief.

> MAKE BACKUP COPIES OF ALL YOUR IMPORTANT PROGRAMS AND FILES

These backup copies are best stored on different data packs or disks. If your first copy is lost or destroyed, the backup saves you the trouble of completely rewriting a program or regathering data. (We kept at least two backups of each chapter of this book.) Extra data packs are available by calling Coleco's toll-free number and at most stores that sell ADAM.

When we use a computer, we save our current work every ten or fifteen minutes. This way, a momentary loss of power or a typing error won't destroy everything. At worst only ten or fifteen minutes' of work disappears. (We once lost about ten pages of this book because of a power problems. It only dimmed the lights for a second, but that was enough. We had forgotten our own rule.)

Suppose you save a program with the name *prog* and later resave it, on

the same data pack with the same name. The first version remains on the data pack as a backup copy. When a backup copy is listed in the directory, its code letter in the directory is a lowercase **a**. If the latest version of *prog* becomes unusable or you need the older version, you can get at (access) the backup copy in two ways. If you don't mind losing the current version you can enter the following commands in direct mode:

```
DELETE prog
RECOVER prog
```

Of course, use the name of your program instead of *prog*.

The first command erases the current version of *prog* and the second converts the backup copy into the current one. (It also changes the code letter listed in the directory back to uppercase.)

To save the current version *and* access the old version, use:

```
RENAME prog, newprog
RECOVER prog
```

The RENAME command changes the name of the current version (to use it you'll need a new name for your program). We often use a number at the end of the new name so *prog1* is the first renamed version, *prog2* the second, and so on.

Relying on these automatic backups is dangerous. They are only accessible while the data pack is usable. To be safe, duplicate important files and programs on *separate* data packs. (On several occasions, we found that something had happened to the directory and the whole data pack was unusable. After using the INIT command, we had good data packs again. However, we had lost all the information that was stored on them.) It is also a good idea to use the SmartWriter to print out hard copies of important files.

Using files is among the most delicate operations in SmartBASIC. This is why it's absolutely essential for this chapter that you not have the earliest

version of SmartBASIC. (The easiest way to test this is to see whether the FLASH command works. It didn't in the oldest version of SmartBASIC.) If you still have this version of SmartBASIC, get the new one before doing any programming using files. (Again, call Coleco's toll-free number to obtain one.)

Self-Check 1

Questions

1. Why are backups important?
2. In a list of the directory contents, what does the code letter **a** mean?
3. What is the different between the two methods of accessing ADAM's built-in backups?

Writing and Reading Data from Files

PRINT tells ADAM to send information out of the computer and into the world. The PR#0 command sends all output to the screen only and the PR#1 command sends it to the screen and the printer. A third PRINT command sets up the data pack drives to get information into or from a particular file. A special code signals that the PRINT statement is not an ordinary print command but a special message to the data pack drives. The code symbol is CHR$(4). To work with a file called ENVELOPE, you enter a statement that looks like:

```
PRINT CHR$(4);"do something with or to ENVELOPE"
```

ADAM has a special program that runs the data pack drives (and does a few other things). This program is called the operating system. (You may have seen the initials OS in an error message. CP/M, which was mentioned

in chapter 1 as an extra cost option for ADAM, is also an operating system.) PRINT CHR$(4) tells ADAM that the rest of the PRINT statement is a command to the operating system.

Imagine you want to work with a file named *phyle* on the data pack. The kind of instructions that we want to send to the drives are familiar to anyone who has used a tape recorder. Here are some:

Tape Recorder Command	Operating System Command
find the song named *phyle*	PRINT CHR$(4);"OPEN *phyle*"
playback *phyle*	PRINT CHR$(4);"READ *phyle*"
record *phyle*	PRINT CHR$(4);"WRITE *phyle*"
add to (the end of) *phyle*	PRINT CHR$(4);"APPEND *phyle*"
we're done with *phyle*	PRINT CHR$(4);"CLOSE *phyle*"

Table 16.1

As you can see, CHR$(4) is used a lot in working with files. It's boring to constantly type it, so most programmers use the assignment:

```
d$ = CHR$(4)
```

early in a file-handling program. This simplifies the commands in table 16.1; for example, the first becomes:

```
PRINT d$;  "OPEN phyle"
```

Now let's use these commands in a simple example:

```
10 REM this program has only one real entry
20 d$ = CHR$(4)
30 PRINT d$;"OPEN phyle"
40 PRINT d$;"WRITE phyle"
```

(continued)

```
50 PRINT "Here's the entry"
60 PRINT "zzz"
70 PRINT d$;"CLOSE phyle"
80 END
```

As we already mentioned, line 20 is there to save us the trouble of writing CHR$(4) in each instruction to the operating system.

Line 30 does three things. First it tells the operating system that we want to use a file named phyle. Next, it adds the name phyle to the directory if it isn't there already. Finally, it sets up a space in ADAM's memory that temporarily holds the information that we want to store in (or get from) phyle. This space is called *buffer.* Without a buffer, each piece of data sent to or from the data pack would be recorded or played back separately. So the tape would spin each time ADAM sends information to or receives it from the data pack. This would really eat up time. For our purposes ADAM has room for two buffers—one to send information to the data pack and one to receive information from the data pack. (This description is not quite accurate but it's a useful one in the context of this discussion. Coleco's technical manual, not yet released at this book's presstime, should have more details.)

A file can have any name using ten or fewer characters and, unlike a variable name, all ten characters matter. It's best to start the file name with a letter, although ADAM allows some exceptions to this rule. No spaces are permitted and certain names are not allowed, for instance ADAM's reserved words. (*Reserved words* are strings that are used by ADAM for special purposes. For example, you can't call a variable *pr* because PR# is a SmartBASIC command.) ADAM recognizes the difference between uppercase and lowercase letters in filenames. For example, as we mentioned in chapter 12, HELLO and hello are different filenames—and only HELLO works as a turnkey program. Operating system commands like OPEN, WRITE, etc., can be either upper- or lowercase.

Line 40 says that ADAM is going to send information to phyle. ADAM "writes" to files to send information and "reads" from files to receive it. The buzzword for a file that is used for writing is a *WRITE file.* A *READ file* is a file that ADAM reads from. The WRITE command also reserves space on the data pack to receive information from ADAM. Unfortunately, ADAM doesn't know how much space to reserve so it automatically reserves all the empty space on the data pack. If you are not careful this can be a major source of problems. You must have ADAM close any file that has been used for writing as soon as you are finished with it—otherwise ADAM will think that the drive holds a full data pack.

EACH TIME THE OPERATING SYSTEM SEES THE WRITE COMMAND, IT GETS READY TO WRITE INFORMATION AT THE BEGINNING OF THE FILE.

(In computerese, we say that the *file pointer is positioned at the beginning of the file.)*

The command "WRITE phyle" says that every PRINT statement from now on will send its output to *phyle*. Also, all input prompts are sent to *phyle* as well. At this point the best rule to follow is:

Don't try to send messages or prompts to the screen when a file is open.

(We'll have more to say about this later.)

Line 50 sends the string "Here's the entry" to the buffer.

Line 60 sends the string "zzz" to the buffer. We do this for the same reason we added a delimiter to the end of DATA statements (chapter 10). It is an aid in determining when there is no more useful information to be read. Other common end-of-file flags are:

no-name	for a name
−1	for a salary
1E38	for a reasonable number

(Recall from chapter 13 that a flag is something holding information about the status of a file or the computer. An end-of-file flag is a signal that there is no more data in the file.) These lines obey the following rule:

ALWAYS WRITE TO A FILE IN SMARTBASIC USING A SEPARATE PRINT COMMAND FOR EACH PIECE OF INFORMATION.

Line 70 says that you are temporarily done using *phyle*. After ADAM sees a CLOSE command there is some housekeeping to do. (*Housekeeping* means the clean-up work that has to be done after some operations have been performed.) First, the CLOSE command tells ADAM that PRINT statements should no longer send their output to the buffer. Second, the buffer is flushed into the data pack. (*Flushing* is the buzzword for cleaning out a buffer and placing its contents somewhere—in this case into the data pack.) You should hear the data pack drive run after the "CLOSE *phyle*" command.

ADAM stored only the two strings—"Here's the entry" and "zzz"—in phyle. The remark statement in line 10 said that there is only one real entry into phyle. This is because the delimiter "zzz" is used as a barrier. When you ask ADAM to read the information in this file, the "zzz"is (or waves) the end-of-file flag to indicate that there is no more data on the file.

Before we show you how to put some serious information into a file, there are a few important things you should know. The operating system *only* recognizes PRINT CHR$(4) as a command if it appears as the first entry in a PRINT statement. You can't use:

```
PRINT "good morning", d$;"OPEN phyle"
```

as a command to the operating system. ADAM would interpret this as an order to display:

```
good morning      ♥OPEN phyle
```

on the screen. The heart is the character printed when a CHR$(4) is sent to the screen. In chapter 8 you saw that CHR$(4) is the code to print CONTROL/D (see also Appendix A). This key combination shows up on the screen as a little heart. ADAM only recognizes operating system commands when the cursor is located in the first column. The commands HTAB(1) or HOME are the easiest ways to reposition the cursor.

Serious problems occur if you stop a program when a file is open. We won't describe all of the horrors that can occur. Some of them are described in later tricks of the trade.

MAKE SURE YOU CLOSE ANY FILE THAT YOU'VE OPENED.

To check whether a file is closed, enter the command CATALOG in direct mode. If a file was left open, then the catalog will show this file as taking up a seemingly impossible amount of space. ADAM *automatically* assigns to any WRITE file left open all the remaining room on the data pack. To free the space on the data pack, the easiest way is to enter, in direct mode, the command STOP (or enter a command that closes the open file by name). STOP closes all open files. Moreover STOP works even when you don't know the name of the file that might be open. The END command *doesn't* close open files. Since the STOP command has this nice property we think STOP is the best way to end a program that works with files.

Trick of the Trade It's curious that STOP, or any error condition that forces a program to end and gives back the prompt], closes all open files and END and CONTROL/C don't. We expect that future versions of SmartBASIC will change this.

There is no point in writing information in a file unless it's used. Here's another important rule about writing and reading files.

NO FILE CAN BE OPEN TO SEND INFORMATION
TO ADAM AND RECEIVE INFORMATION
FROM ADAM AT THE SAME TIME.

The method we will use to receive information from files works when the file has a flag or you know the exact number of entries. Other methods are sometimes necessary but they must be used with care. For that reason we've put them into tricks of the trade (see the next section).

To read information previously stored in a file you must:

1. Open the file using its name (first make certain that the cursor is in the first column on the screen).

2. Issue a READ command using the file name.

Here's a short demo program to read back the information from our file named *phyle*:

```
 10 REM this program reads phyle and prints its contents
 20 d$ = CHR$(4)
 30 PRINT d$;"OPEN phyle"
 40 PRINT d$;"READ phyle"
 50 INPUT a$
 60 IF a$ = "zzz" THEN 90
 70 PRINT a$
 80 GOTO 50
 90 HTAB(1)
100 PRINT d$;"CLOSE phyle"
110 STOP
```

This is the obvious program to write. By opening *phyle* as a READ file, we play back its contents. Lines 50 through 80 contain a loop with a test that plays back the real contents of *phyle* and also prints them. Line 90 is very important. Notice how it moves the cursor back to the first column *before* issuing the CLOSE command.

Self-Check 2

Questions

(In the following questions, use ADAM to see what happens when you don't follow the rules. In the answers, we'll show you how to clear up any mess this might cause.)

1. What happens if you try to write to a file that hasn't been opened?
2. What happens to a file that doesn't get closed after it has been opened? How does this affect future use of the data pack?
3. What happens if you remove a data pack, containing an open file from the data pack drive, before the file is closed?
4. How can ADAM's word processor be used to check the contents of a file?

Using Data Files with Programs

The paintbrush programs (on pages 176 and 266) are good examples of programs that can be improved using files. With the paintbrush programs, it takes time to paint the graphics screen. When you are finished painting and turn ADAM off, your work goes down the drain—unless you store the results in a file. Adding a small module to the original paintbrush program lets you store the color codes for the whole graphics screen in a file. Afterwards, a short program will recreate a duplicate of the original. We'll show you how to do this for one of the paintbrush programs and leave it to you to change the other. Both versions can use the same repainting program.

The modified paintbrush program is the first serious program using files. It's worth stopping and thinking about what would happen if the program didn't work. There are times that things we do cause ADAM to send out an error message and stop the program. Why care if ADAM sends an error message and stops the program? The answer is simple and sometimes deadly! When ADAM sends an error message, it is telling you that it has lost control of the situation. Strange things might get written onto the files or information you need may never reach them.

To prevent this you'll need the special error trapping command:

```
ONERR GOTO ____
```

This command stops ADAM from bombing a program when an error occurs. It overrules ADAM's normal procedures. Whenever an error is detected ADAM goes to the line indicated instead of bombing the program. Since there are many different kinds of errors you have to know which error occurred. SmartBASIC has another command:

```
ERRNUM(1)
```

which gives you the code number for many different errors. For example I/O ERROR is error code 8. (Recall the I/O ERROR is computerese for input-output error. This means that ADAM is having problems getting data to or from a data pack.) The most common reasons for this error are:

1. The data pack is inserted backwards, inserted incorrectly, or not inserted at all.

2. The tape head is dirty or misaligned (a much more serious problem).

By inserting the lines:

```
e% = ERRNUM(1)
IF e% = 8 THEN .....
```

we can test whether the error was an I/O ERROR. The THEN clause can contain instructions to check the data pack and then go back to the filing operation.

Suppose we mistakenly place a full data pack in the drive. We can test if the data pack is full using an error trap. Error code 9 is NO MORE ROOM. There are quite a few error codes and a complete list is given in Appendix B.

The ONERR command isn't a cure-all; be very careful when using it. You are taking away one of the safety nets that SmartBASIC provides to ensure that your programs operate correctly. Using ONERR doesn't hurt ADAM, but your program can give weird results instead of bombing. For example, when ADAM detects an error condition, it closes files before it prints the error message. When you use ONERR_____GOTO, to prevent errors from writing weird thing into files you'll need to use another method of closing all open files. We rarely use ONERR_____GOTOs in programs using files unless they send control to an error trap that closes all files. It doesn't matter if you use a CLOSE command, a STOP command, or some other method. Just be sure that the error trapping module closes all open files before the program ends.

We avoid the ONERR_____GOTO as much as possible—and after using it, we turn it off as soon as possible. For example, it is not needed after

ADAM has closed the files that we were working with. It is turned off by inserting:

```
CLRERR
```

in any line after the ONERR GOTO command.

Now to the modified paintbrush program. We'll modify the program on page 176 so that it copies the screen data into a file. First we have to change two lines in the program. When you want to say that the painting is done, enter e in response to the input prompt on line 550. The variable *ch*$ gets the value e. Line 600 sends control to line 140 which is the TEXT command and line 150 is the END statement. We first make the following changes:

```
150 STOP
600 IF ch$ = "e" THEN GOSUB 2500:  GOTO 140
```

Line 600 creates a detour to the filing subroutine. The filing subroutine is:

```
2495 REM
2496 REM
2497 REM ************************************************
2498 REM  module to record a painting
2499 REM ************************************************
2500 INPUT "Do you want to file this painting (enter y or n)"
; yn$
2510 IF yn$ <> "Y" AND yn$ <> "y" THEN RETURN
2520 d$ = CHR$(4): op$ = "open ": wr$ = "write ": cl$ =
"close "
2530 HOME:INPUT "What is the name of the file you want to
use ";f$
2540 ONERR GOTO 3000
2550 PRINT: PRINT d$ + op$ + f$: PRINT d$ + wr$ + f$
2560 FOR col = 0 TO 39
2570 FOR row = 0 TO 39
2580 PRINT SCRN(col,row)
2590 NEXT row: NEXT col
2600 PRINT d$ + cl$ + f$
2610 CLRERR
```

(continued)

```
2620 RETURN
2995 REM
2996 REM
2997 REM ************************************************
2998 REM  file handling error trap
2999 REM ************************************************
3000 e% = ERRNUM(1)
3010 IF e% = 9 THEN PRINT "NO MORE ROOM. CHANGE THE DATA
PACK": GOTO 3030
3020 IF e% = 8 THEN PRINT "I/O ERROR. CHECK THE DATA PACK"
3030 PRINT "HIT C TO CONTINUE OR S TO STOP": GET zz$
3030 IF zz$ = "c" OR zz$ = "C" THEN 2550
3040 PRINT: PRINT d$ + cl$ + f$
3050 GOTO 140
```

These modules are not very exciting. The recording module goes systematically through every box in the low-res graphics screen and files the color for each block. The only new feature in this module is that there is no flag. When ADAM reads this file, the program will have nested loops with the same starting and test values. ADAM knows exactly how much data to expect, so a flag serves no purpose.

We've used one new trick here to save typing. Line 2520 contains assignment statements for most of the command words to handle files. For example, on line 2540, the PRINT statements translates into:

```
PRINT CHR$(4);"open ";f$
```

and:

```
PRINT CHR$(4);"write ";f$
```

because, in PRINT statements, the ; concatenates the strings *d*$, *op*$, and *f*$. The name of the file where the painting is to be stored is contained in *f*$. When using this program pick a name that reminds you of the painting. To name the file PICASSO, answer PICASSO in response to the prompt from line 2530. After this line 2540 is translated by ADAM to:

```
PRINT CHR$(4);" open PICASSO"
```

We're making a fuss about this trick because many computers won't allow you to specify the names of files using INPUT or DATA statements. On those machines, file names must be built into the programs. SmartBASIC allows you this extra flexibility in the most natural way, namely, the program changes the string sent to the operating system. There is one caution you must observe in using this trick—the space following the words *open, read, write,* etc., in line 2520 cannot be left out.

Notice how we stored *all* the information in an array before writing the information to a file. This simplifies many file handling operations. When you use this simplified method you are restricted to using short files. For our purposes, a file is short if all its information fits into an array in ADAM's memory—that's about twenty pages, so it's not really so short. (Remember that the program must also fit in memory.)

The various HTAB(1) commands and empty PRINT statements scattered through the program may look strange. They would be in any kind of program other than a file handling program. We're just following the two basic rules about filing:

1. The cursor must be in the first column.

2. The previous PRINT statement must end in a carriage return.

While we're on the subject of being careful, GET can also interfere with operating system commands that follow.

> IF YOU USE A GET COMMAND FOLLOWED BY AN OPERATING SYSTEM COMMAND, SEPARATE THE TWO BY AN EMPTY PRINT STATEMENT.

Recording the painting is useless without a playback program. Here's one way to do that:

```
10 REM playing back a painting
20 d$ = CHR$(4): op$ = "open ": rd$ = "read ": cl$ = "close "
30 HOME: INPUT "Which file contains the painting? "; f$
40 GR
50 HTAB(1): PRINT d$ + op$ + f$: PRINT d$ + rd$ + f$
60 FOR col = 0 TO 39
```

(continued)

```
70 FOR row = 0 TO 39
80 INPUT x%: COLOR = x%: PLOT col,row
90 NEXT row
100 NEXT col
110 HTAB(1): PRINT d$ + cl$ + f$
120 HOME: PRINT "Press any key to end the program"
130 GET a$: TEXT
140 STOP
```

This program is short so there is no need to break it up into modules or use subroutine calls. Error trapping is also useless because it is easier to correct the problems and rerun the program. There is no need for an end-of-file flag because there are always 1600 (= 40 * 40) color codes in the file. Again notice that the HTAB(1) command on line 110 repositions the cursor to the first column so that ADAM will understand the next instruction as an operating system command.

Trick of the Trade One of the dangers of error traps involves CONTROL/C. This method of stopping a program is caught by any error trap using the ONERR command. In other words, CONTROL/C won't stop a program using the ONERR command directly. To make CONTROL/C work the error trap must include some way to stop the processing. (If you forget to do this, you're stuck. The only way to end the program is to hit the reset button and reload SmartBASIC. This can leave files open on your data pack. Be sure to check and close them if necessary.) When we develop a long program, our error traps close all open files, tells us the error code number, and then ends the program.

An added use of ONERR is to read back information from a file when you don't know if the file has a flag. The following subroutine uses ONERR_____GOTO to read a file:

```
997 REM ****************************************
998 REM  reading files using ONERR GOTO
999 REM ****************************************
1000 d$ = CHR$(4)
1010 ONERR GOTO 10000
1020 PRINT: PRINT d$;"OPEN (        )"
1030 PRINT d$;"READ (        )"
1040 INPUT a$
1050 GOTO 1040
1060 HTAB(1): PRINT d$; "CLOSE (        )"
1070 CLRERR: RETURN
10000 a% = ERRNUM(1)
```

(continued)

```
10010 IF a% = 5 THEN 1060
10020 PRINT "FATAL ERROR"
10030 STOP
```

You must fill in the file names instead of the parentheses in lines 1020, 1030, and 1060. This module sets up an infinite loop to read the data. (Line 1040 could be replaced by a subroutine call that reads and processes the information.) Eventually ADAM runs out of DATA, sends the OUT OF DATA message (error code #5), and goes to the error trapping module.

Self-Check 3

A. Questions

1. Suppose ERRNUM(1) = 42. What error does this indicate? (Consult the table at the back of the book.)
2. What does the CLRERR command do?
3. What are the advantages and disadvantages of the ONERR command?

B. Program

Write a program that will store the names of your friends in a string array. Write the information to a file. Now write a program that prints a personalized letter to each of them.

Coded Messages

Files are usually created by a program and read by the same or other programs. You can also move data from one ADAM computer to another by carrying it in a data pack. We want to show you how this data can be arranged or encoded so that only people having the right decoding program can understand the information in the file. You can also use the same program to keep a private diary. Before looking at the program, it's worthwhile seeing how codes work and are used.

Coded messages used to be physically carried from the sender to the

recipient. Both parties had copies of a codebook or key which was used to translate uncoded material into code and vice versa. After radio was developed, it became more convenient (and safer) to broadcast coded messages. Now you can try to detect messages in their coded form and try to break the codes, usually using computers. Coding methods are much more sophisticated now and often use random numbers. As long as the sender and the recipient use the same table of random numbers, the message is easily decoded. However, this type of code is difficult to break. Our encoder program uses the random number generator in ADAM to code a message. To make the code more difficult to break, the sender chooses a seed (see chapter 9). This acts as the key to the code.

Here's the program to set up a coded file:

```
10 REM  program encode
20 DIM a%(8000)
30 HOME: d$ = CHR$(4): op$ = "open ":wr$ = "write ": cl$ =
"close "
40 GOSUB 500: REM   set the key for the code
50 GOSUB 1000: REM  name the file to hold the message
60 GOSUB 1500: REM  enter and file the message
70 HOME
80 STOP
495 REM
496 REM
497 REM ******************************************************
498 REM  code key module
499 REM ******************************************************
500 INPUT "Enter the key for the code (a positive integer) ";
ky%
510 x = RND(-ky%)
520 RETURN
995 REM
996 REM
997 REM ******************************************************
998 REM  file naming module
999 REM ******************************************************
1000 INPUT "What is the name of the file   you want to hold
the message   (don't use spaces, commas or    quotes)? "; f$
1010 PRINT "You can start typing."
1020 PRINT "Hit SmartKey I to stop.": PRINT
1030 RETURN
1495 REM
1496 REM
1497 REM ******************************************************
1498 REM  data gathering and filing module
1499 REM ******************************************************
1500 FOR i = 1 TO 8000
1510 GET b$: PRINT b$;
```

(continued)

```
1520 IF ASC(b$) = 129 THEN 1550
1530 a%(i) = ASC(b$) + INT(100*RND(1))
1540 NEXT i
1550 ONERR GOTO 2000
1560 PRINT: PRINT d$ + op$ + f$
1570 PRINT d$ + wr$ + f$
1580 PRINT ky%
1590 FOR j = 1 TO i-1: PRINT a%(j): NEXT j
1600 b% = -1: PRINT b%
1610 HTAB(1): PRINT d$ + cl$ + f$
1620 CLRERR
1630 RETURN
1995 REM
1996 REM
1997 REM ************************************************
1998 REM  file handling error trap
1999 REM ************************************************
2000 e% = ERRNUM(1)
2010 IF e% = 9 THEN PRINT "NO MORE ROOM. CHANGE THE DATA
PACK": GOTO 2030
2020 IF e% = 8 THEN PRINT "I/O ERROR. CHECK THE DATA PACK"
2030 HOME: "HIT C TO CONTINUE OR A TO ABORT": GET zz$
2040 IF zz$ = "c" OR zz$ = "C" THEN 1560
2050 PRINT d$ + cl$ + f$
2060 GOTO 70
```

The main module (lines 10 through 80) is a bunch of subroutine calls together with the assignment statements for the commands used in file handling.

Line 20 sets a limit on the length of the message. It can't be longer than 8000 bytes or characters (about eight double-spaced pages). ADAM probably could handle a 10-page message (about 10,000 bytes) because the message is stored in an integer list (see chapter 11).

The key to the code is entered in the code key module. The file naming module allows you to name the file.

The data gathering and filing module does most of the work in this program. After each character is entered, ADAM computes its ASCII code and adds a random number to it. The person using the program was previously told (line 1020) to hit SmartKey I at the end of the message. SmartKey I sends ASCII code 129 to ADAM (see Appendix A). When ADAM sees that code, it first sets up an error trap. The best-written programs can have problems with file-handling—the data pack may be badly positioned in the drive or damaged. Either causes an I/O ERROR message.

Another typical problem is that the data pack or, more likely, its catalog, is full. The error trapping module built into this encoding program lets

you check or change the data pack. After checking this, you can restart the filing process without losing everything you've entered.

Next, ADAM opens the file you named using the variable *f*$ and then it begins filing the coded message. We programmed ADAM to write a −1 at the end of the file. The −1 acts as an end-of-file flag. Then ADAM closes the file and the program ends. Notice that every route out of the program goes past a CLOSE statement.

The error trap has some interesting features. Once an error trap has been set up, all errors force ADAM to follow the instructions in the trap. The only types of errors that we can correct are the I/O ERROR and NO MORE ROOM. Every other error goes directly through the error trap and then ADAM goes to the STOP statement. The message *in memory* isn't destroyed by I/O or OUT OF DATA error conditions. After the operator fixes the problem, ADAM starts refiling the whole message. Another command could be used here: RESUME. This command tells ADAM to reprocess the command that sent it into the error trap. For example, line 2040 could be changed to:

```
2040 IF zz$ = "c" OR zz$ = "C" THEN RESUME
```

It's more dangerous to use RESUME than to refile the entire message. The best cure for the error condition is probably to use a different data pack, but the open file is on the first data pack. RESUME won't work under these circumstances. RESUME should be used when only part of the data is in memory. This way, if you can't refile the data from the beginning, you try to save whatever you can.

A program to encode a message is useless without one to decode it. A decoding program is easier to write than an encoding one because no safety measures are needed. For example, if ADAM sends out an I/O ERROR message, you can always run the program again without losing any information. If the encoding program bombed, all your typing would have been wasted. Moreover, because decoding reads from a file rather than writing to it, decoding doesn't add information to the file. This way you never have to worry about running out of room on the data pack when using the decoding program. The main operations in the decoding program are the subroutine calls in lines 50 to 80.

```
10 REM  program decode
20 REM  decodes a message from program encode
30 HOME: d$ = CHR$(4): de$ = "delete ": op$ = "open ":
rd$ = "read ": cl$ = "close "
40 DIM a% (8000)
50 GOSUB 500: REM  startup module
60 GOSUB 1000: REM read module
70 GOSUB 1500: REM close file and print message
80 GOSUB 2000: REM erase module
90 STOP
495 REM
496 REM
497 REM ******************************************************
498 REM  startup module
499 REM ******************************************************
500 INPUT "What is the name of the file containing the
message? "; f$
510 INPUT "Do you want to print the message (enter y or n) ";
yn$
520 HTAB(1): PRINT d$ + op$ + f$
530 PRINT d$ + rd$ + f$
540 RETURN
995 REM
996 REM
997 REM ******************************************************
998 REM  read module
999 REM ******************************************************
1000 INPUT ky%: REM  get the key
1010 x = RND(-ky%)
1020 FOR i = 1 TO 8000
1030 INPUT b%
1040 IF b% = -1 THEN 1070
1050 a%(i) = b% - INT(100*RND(1))
1060 NEXT i
1070 RETURN
1495 REM
1496 REM
1497 REM ******************************************************
1498 REM  file closing and printing module
1499 REM ******************************************************
1500 HTAB(1): PRINT d$ + cl$ + f$
1510 IF yn$ = "Y" OR yn$ = "y" THEN PR#1
1520 HOME: PRINT: PRINT: PRINT
1530 FOR j = 1 TO i-1
1540 PRINT CHR$(a%(j));
1550 NEXT j
1560 PR#0
1570 RETURN
1995 REM
```

(continued)

```
1996 REM
1997 REM ************************************************
1998 REM  erase module
1999 REM ************************************************
2000 PRINT: INPUT "Do you want to erase the message from the
data pack (enter y or n)? "; yn$
2010 IF yn$ = "Y" OR yn$ = "y" THEN HTAB(1):
PRINT d$ + de$ + f$
2020 RETURN
```

Some of the choices we made in writing this program are worth noting. To use the startup module, the person decoding the message must know the name of the file containing it. (Of course, the recipient *could* try to decode every message on the data pack.) We used the same trick as in the encode program to tell the operating system that the file named *f*$ should be opened and read from. The READ module could also ask the recipient to enter the code key and compare it with the key in the file. This gives one more level of security. We didn't do this because the program is long enough as is. Lines 1020 to 1060 tell ADAM to read and decode the message (into ASCII) until the end-of-file flag is found.

The ASCII codes for the message are stored in the list *a*%. Since the message is stored in a list, the recipient has many options for displaying or printing the message. An expanded program would include a printing menu. After some method to display or print was entered, an ON-GOSUB command would move to the required module. The file closing and printing module does exactly what its name indicates. The recipient chooses either to display the message on the screen or print it. Since the list *a*% contains the ASCII codes for the characters in the message, ADAM must translate the codes back to characters before printing them. The last module gives the recipient the choice of saving the message on the data pack or erasing it.

Self-Check 4

A. Questions

1. We showed you a trick for using a file name that you've entered in OPEN, READ, and WRITE commands. Do the same thing for DELETE and RENAME.
2. In the encode and decode programs, we used the integer variables *ky*% and *a*%. What is the advantage of doing this? Did we really need to do it for both variables?

3. The end-of-file flag in the message file is −1. Why is this an absurd value for the data stored in the file?

B. Programs

1. Change the decode program so the recipient must enter the correct code key or the bell sounds and the program ends with a nasty message about reading other people's private messages.
2. Change the decode program so the print is formatted, as in chapter 13. Printed words shouldn't be split at the end of a printed line.
3. Make up a menu for the decode program so the recipient can choose the type of output he or she wants. Use subroutines and an ON-GOSUB to implement the choice.
4. The encoding line (1530) and the decoding line (1050) must be matched, but there are many other ways of coding messages. Try two different methods in the encode and decode programs.

How to Add Information to a File

Unless we can change the contents of files, the only advantage they have over DATA statements is that they can hold the output of a program. We've mentioned several times that SmartBASIC was designed so that Applesoft programs would run on ADAM with little or no change (see Appendix E for a list of the major differences). Applesoft's way of writing to files has many do's and don'ts. The current version of SmartBASIC does a few things differently than Applesoft. These changes may seem quite minor but they either limit you to working with short files or test if you have the patience of a saint. We can't believe that Coleco will permit these silly limitations to remain in SmartBASIC. However, we have to write about the version we have, not the version we expect. (We hope you'll learn enough about files to use the version we hope will appear.)

Where the Problems Come From

Suppose you want to add something to the beginning of a file. The data entries in the file are in order. The first position in the file is already occupied. If you use the WRITE command you'll write over the current contents of the file. Luckily there's a trick. First you write the new data to a different file, called a *temporary file.* Then you copy the contents of the original file onto the end of the temporary file. SmartBASIC has a command for copying onto the end of a file. The command is APPEND and it is used in statements like:

```
PRINT d$; "APPEND phyle"
```

In SmartBASIC, like Applesoft, APPEND also opens the file so no OPEN statement is needed before appending. APPEND sends information to the point on the data pack after the last entry in phyle. In computerese, *APPEND sets the pointer after the last entry in phyle.* In pseudocode, here's the obvious way to append information from the file *fiel* to the end of *phyle:*

```
First output whatever you want to phyle
give the command to append to phyle
open fiel and read from fiel
begin a loop
      read an entry from fiel (using INPUT or GET)
      write it into phyle (using PRINT)
next
close the open files
```

First the good news. With this method you don't have to worry about the length of *fiel.* Its entries pass through ADAM one at a time. Now the bad news. *This method doesn't work.* It doesn't work in SmartBASIC or in Applesoft. The reason is simple. When a file is opened for writing or appending, *any* INPUT statement messes up the filing. It doesn't matter whether the INPUT comes from the keyboard or another file, it's still messed up. Applesoft uses a trick to get around the problem. Any operating system command, like:

```
PRINT d$; "OPEN newfile"
```

or even the command:

```
PRINT d$
```

which says nothing, cancels all previous READ, WRITE, and APPEND commands *without closing the file.* That means that an INPUT command has nothing to interfere with. After the INPUT statement, we have to tell ADAM to start reading from *fiel* by issuing the commands:

```
PRINT d$; "READ FIEL"
```

To write *a$* to *phyle*, we should use the commands:

```
PRINT d$; "WRITE phyle"
PRINT a$
```

This works in Applesoft but not in SmartBASIC. In Applesoft, the OPEN command sets the pointer to the first entry, so the pointer is still pointed at the last entry and *a$* is written where we want. In SmartBASIC, the WRITE command sets the pointer to the first entry. So the commands we just wrote write the contents of *a$ over the first entry in phyle.* This method destroys SmartBASIC files.

Using APPEND instead of WRITE in the last example creates other problems. If you use the commands:

```
PRINT d$; "CLOSE phyle"
INPUT a$
PRINT d$; "APPEND phyle"
PRINT a$
```

This will cause *a*$ to wind up as the last entry in phyle—exactly where we want it. But the data pack may spin a few minutes before we can enter the next piece of information. We find the wait intolerable.

You could also leave off the CLOSE part and use only PRINT *d*$. This may work once (that is, for one value of *a*$). When you use this command, ADAM sets aside another buffer for appending to phyle. After you've done this twice, ADAM runs out of buffers.

The moral of our story is that, at present, SmartBASIC can't really handle long files (that is, files too long to hold in memory). If Coleco comes up with an improved version that stays compatible with Applesoft, the version will obey the rules we just mentioned and you'll have no trouble simultaneously reading and writing to files.

You can't use any of the tricks we mentioned above with the current version of SmartBASIC. If you restrict yourself to relatively short files (six pages or so), all these problems become manageable through programming. The idea is simple. Anything you want to INPUT from a file, put somewhere in ADAM's memory. Then do what you want. Finally send all the information to the data pack. Here's one example.

Let's take a file that has string entries and add entries to the beginning of the file.

```
10 REM adding entries to the beginning of a short file
20 d$ = CHR$(4): qu$ = CHR$(34)
30 op$ = "open ": rd$ = "read ": wr$ = "write ": cl$ =
"close ": del$ = "delete ": ren$ = "rename "
40 INPUT "What is the name of the file you want to change? ";
f$
50 HTAB(1): PRINT d$+op$+f$: PRINT d$+rd$+f$
60 count = 0
70 count = count+1
80 INPUT a$
90 IF a$ = "zzz" THEN 110
100 GOTO 70
110 HTAB(1): PRINT d$+cl$+f$
120 DIM a$(count)
130 PRINT d$+op$+f$: PRINT d$+rd$+f$
140 FOR i = 1 TO count
150 INPUT a$(i)
160 NEXT i
170 HTAB(1): PRINT d$+cl$+f$
180 PRINT d$+del$+f$: REM we don't need the old file anymore
190 DIM b$(FRE(0)/8)
200 kount = 0
```

(continued)

```
210 PRINT "Enter the strings one at a time."
220 PRINT "Press return after each entry."
230 PRINT "Use zzz as the last entry"
240 INPUT b$(kount)
250 IF b$(count) = "zzz" THEN 280
260 kount = kount + 1
270 GOTO 240
280 HTAB 1: PRINT d$+op$+f$: PRINT d$+wr$+f$
290 FOR i = 1 TO kount: PRINT b$(kount): NEXT
300 FOR i = 1 TO count: PRINT a$(count): NEXT
310 PRINT d$+cl$+f$
320 STOP
```

There are several interesting things about this program. As soon as we took all the data from file *f*$, we threw it away (deleted it may sound fancier). So the name was free and we could use it for the new file created by the program. The two counter variables, *count* and *kount*, act differently. (See the self-check below.) We used the command FRE(0), to see how much memory was left after the contents of the original file were read. This gives us an idea of the room left for the additional strings. We're assuming that the strings will average less than eight characters. An even better idea would be to trap the out-of-memory error message. The error trapping module should pass control to line 280 to file as much as we can. Finally notice that the HTAB(1) command was used before trying to close *any* READ file.

Self-Check 5

A. Questions

1. What are the differences between the commands WRITE and APPEND?
2. In the program on page 432, how do the two different counters work?
3. What changes in the program would tell ADAM to add the new material to the end of the file rather than the beginning?

B. Programs

1. Write an error trap for the program on page 432 as described in the preceding text.

2. Write a program to store the names and prices of all items in your house that are worth more than $500.
3. Write a program that reads the file created by the previous program and print the items and their prices using the printer. (These two programs keep an inventory of the valuables in your house. An inventory is often useful when you have to make an insurance claim.)

Some Suggestions on When and How to Use Files

In the modified paintbrush programs, files are used to store the output of the programs. The other important use of files is to store the input for one or more programs. For example, any data kept in DATA statements can also be stored in a file. Complicated DATA (and INPUT) statements are often useful, but they shouldn't be used for data that is changed or updated frequently. For example, the program on page 190 prints the word ADAM using large block letters. It is incredibly complicated PRINT statements. Suppose someone distributed a file containing the data needed to print all the letters in this way. After that, the program is very simple—essentially just read the DATA and issue some PRINT commands. Giant-sized letters are fun to have and may be useful (for example, to print banners). It's silly to enter them into just one program as DATA statements, or worse, as horrid PRINT commands. (Before you complain, remember at that point in the book, you didn't know about DATA statements.) A file containing the whole alphabet would contain over a thousand separate codes. (If anyone wants to create such a file and distribute it, we would like to have a copy.)

Finally, data that must be changed frequently should be kept in files rather than in DATA statements. Modifying DATA statements means changing the program itself (because DATA statements are part of the program). It's not a good idea to frequently change a working program—you're likely to mess it up.

To summarize, here are some rules of thumb for when to use data files:

1. If rerunning a program takes a lot of time or effort, the output of the program should be kept in a file. Instead of rerunning the program, you just ask ADAM to print, plot, or display the output.
2. If the input data used in running a program needs frequent change, that data should be kept in a file. You have to write another program to get the data into the file or to update the data. (A file can be changed or created using ADAM's word processor.)
3. If data is useful for several different programs, that data should be stored in a file.

We've already shown you how to SAVE, LOAD, RENAME, DELETE, and RECOVER programs. There are several other operating system commands to manipulate program files. These commands are used in direct mode by entering the command (in the format shown in table 16.2) or in a program (see below).

Additional Operating System Commands

Optional specifications are enclosed in brackets. If a comma is also in the brackets and the specification is not used, the comma should not be used.

INIT — erases all (unhidden) entries in the directory of a data pack. The effect is to make the data pack look brand new. The command is very dangerous because there is no safety feature to keep you from erasing important programs and data.* (If you buy blank data packs for each member of your family, you can use each one's name as the *volumename*. This labels each data pack.)

format: (in direct mode) INIT *volumename*[,D#]

LOCK	blocks the command DELETE from having any effect on a file. (Files that are locked appear with a * in the directory.)
	format: (in direct mode) LOCK *filename*[,D#]
UNLOCK	turns off the command LOCK. After being UNLOCKED, a LOCKED file can be deleted.
	format: (in direct mode) UNLOCK *filename*[,D#]
RUN	loads and starts the processing of a program on the data pack.
	format: (in direct mode) RUN *filename*[,D#]

*The file containing SmartBASIC is hidden. It does not appear in the directory of the data pack that contains it.

Note: The drive number, D#, is always optional. If the drive number is omitted, the command works with the default drive (the left data pack drive). If you buy another drive, this would be D2 in these commands. The disk drive will be number three. All the other commands such as RENAME or DELETE also require a drive number to access the second digital data pack. (If SmartBASIC is loaded from the disk drive, drive 3 is the default drive.)

Table 16.2

These commands can be used in a program. The rule is, an operating system command can be used in a program in a line like:

```
PRINT CHR$(4);  "name of command filename"
```

INIT isn't useful in programming mode. The other commands are. For example, in the program on page 432 we used DELETE to get rid of a file and save different information under the same name. LOCK can be added to the paintbrush recording program to prevent the data file from being erased. Add the line:

```
PRINT CHR$(4);"LOCK PICASSO"
```

and then your painting is safe.

The CATALOG command is often used to allow someone to check the contents of a data pack before trying to write data on it. Examples of this can be found in the programs in the Afterword.

The most interesting command listed in table 16.2 is the old command RUN. Suppose we have a program running and want ADAM to start running another program, say one named *test3*? The program line:

```
PRINT CHR$(4);"run test3"
```

tells ADAM to load *test3* and then start running it. This erases the program that was in ADAM's memory.

The idea of letting programs call other programs has been around almost as long as computers. The first microcomputers (and also the first computers) had very little memory—some as little as 2K (2000 bytes). Long programs couldn't be stored in memory. To overcome this problem, there was a command called CHAIN. (*Chaining* is the buzzword for programs running one after another with no action on your part.) SmartBASIC doesn't have the CHAIN command, but it is easily faked using RUN. There are other reasons for chaining programs besides running out of memory. If you want a program to do two things and have already written separate programs that do each of them, you can chain the second program to the first. Here's an example. Earlier in this chapter, we showed you two programs called *encode* and *decode*. Once a message is encoded, you might want to check it automatically. Add:

```
74 INPUT "Do you want to check your message?
   (Enter y or n )"; yn$
77 IF yn$ = "y" THEN PRINT CHR$(4); "run
   decode"
```

to encode. These extra lines allow decode to run automatically. (If decode is not on the data pack in the (default) drive, the program bombs.)

When a new program is loaded, memory is cleared, so no data is automatically passed from the first program to the second. To transfer information from the first program to the second, you must store it in a separate file on the data pack. The new program calls that file to get the data.

Tricks of the Trade This use of the RUN command causes some problems in the current version of SmartBASIC. This explains why mistakenly typing:

```
RUN ]
```

and hitting RETURN makes the program in memory disappear. (We warned you about this in chapter 4.) ADAM thinks that you want to run a program called]. The current program in memory is erased. Also ADAM can't find a program named]|. Unfortunately, the] key is close to the RETURN key, so it's easy to hit them both when you only want to press RETURN after typing RUN. The only advice we can give is to be careful and keep backups.

ADAM can work with four different types of files. We've discussed two of them—program files and ASCII files. The other two types are random access files and binary files. Both are data files. We won't discuss any of the specialized commands that are used only with these types of files, but the next section of this chapter summarizes all the filing commands currently available in SmartBASIC in a table.

Random access files have entries of fixed size. Each entry in a random access file is found by knowing its position in the file. When the operating system looks for a specific entry, the data pack drive (or floppy disk drive) moves directly to the correct position without first reading every entry that precedes it. (ASCII files—the kind used in this chapter—are often called sequential files because the entries must be read in order.) To find a particular entry on a data pack, ADAM has to spin the tape to the proper position—it can't go directly to an entry. This means there is no significant speed advantage in using random access files with data pack drives. (There are other, more technical reasons that random access files work slowly with the data pack operating system.) That's the main reason we haven't included a discussion of these files. If you have the ADAM disk drive, random access files will work much faster than sequential files.

Binary files are completely different. They are sequential files written directly in the language used by the central processor. You can think of

them as files in machine language. Since we aren't showing you machine language—or even the symbols used in that language—it's impossible to give a decent description of binary files. The most important advantage of binary files is speed. The speed advantage is in using the data *not* in finding the entries in the file. For example, binary files can hold shape tables. You can also store machine language programs in a binary file. Using some of ADAM's most exciting features will require mastering binary files. This information will (eventually) be found in more advanced books than this.

Tricks of the Trade Programs that don't use files are often debugged by studying what they do. We showed you two tools to see how a program works. PRINT statements inserted directly in the program can tell ADAM to display values of variables. You can also use the TRACE command to show which statements are being processed. New techniques are needed to debug programs that use files. Suppose we try to move information to a file. There is no way to see whether it is actually recorded on the data pack. However, SmartBASIC has a special command, MON, which tells ADAM to display information (including listings of programs) as it moves between the computer and the data pack. Unfortunately, MON only can tell you that information is going to the data pack. It can't say whether it ever reaches its destination. If MON says that information was on its way but you later find that it never got there, then ADAM is broken. (In computerese, we say there was a *hardware failure.*)

MON is an abbreviation of the word *monitor* meaning "to keep track of." When using MON, you have to tell ADAM exactly what information you want to see. Table 16.3 lists the different options.

Command	Action
MON C	displays the commands going to the data pack.
MON L	displays the contents of a SmartBASIC program file as it is loaded (moved from the data pack into ADAM's memory).
MON I	displays the input from the data pack as it is moved into ADAM's memory.
MON O	displays the output from ADAM's computer as it moves to the data pack.

Table 16.3

MON is usually used in direct mode. More than one MON option can be turned at the same time. For example, you can enter:

```
MON I,O
```

to display the input and output, or:

```
MON I,O,C,L
```

to monitor all four possibilities. If a program isn't in memory, it has to be loaded before it can be listed. These two operations, loading and listing, can be combined. If you enter:

```
MON L:  LOAD proname
```

ADAM will display the lines in the program named *proname* as they go into memory. This saves a little time.

The command to turn MON off is NOMON. As with MON, at least one of the letters I, O, C, or L must follow this command. ADAM only stops monitoring the operations that are turned off. For example, if MON I, O, or C is on (input, output and commands to the data pack are displayed), entering NOMON C only turns off the monitoring of commands. Input and output will still be monitored.

To watch a file-handling program in progess, MON is usually used with TRACE. MON shows you information passing between ADAM and the data pack drives and TRACE tells you which line is being processed. When you use them together, there is one big problem. TRACE blocks data from going to and coming from the data pack. Each time input is expected from the data pack, a ? appears on the screen telling you that ADAM is expecting input. When a file is being read, all requests for input are directed to the data pack. In other words, the data pack is supposed to answer the input request. TRACE stops it from passing any information to ADAM, so you have to answer the request yourself. When some data is entered, the program will continue. For example, if we turn on TRACE and then run the

program on page 421 (playback a painting), each INPUT command will tell ADAM to print a ? on the screen. However, no information will come from the data pack and ADAM will keep waiting for some input. To trace the operation of the program, you must enter some number between 0 and 15 in response to the prompt. Then ADAM will continue processing the program.

There is another trick for getting around this problem. The commands TRACE and NOTRACE can be used as instructions in a program. When you TRACE a program and come to lines that direct information to or from the operating system or the data pack drive, turn off TRACE. After the file handling is done, turn TRACE on again.

For now, we suggest writing programs involving files by setting up an array or list, filling it, and then writing all the information to a sequential file on ADAM's data packs in a loop that doesn't do anything else. To write a program taking information from data packs, do the opposite. Set up a list or an array and fill it *before* starting to manipulate it. By the time you're comfortable with this method, we expect that a new version of SmartBASIC—or SmartBASIC II, which has already been announced—will be available. Hopefully the new version will enable ADAM to interact with someone while files are open. If this improvement follows the Applesoft rules, you can use the techniques shown in this chapter to write this type of program.

The next section is a complete list of the file handling commands as they currently exist in SmartBASIC. Many of them are not currently useful but they may be in the future.

SmartBASIC's File Handling Commands

This section contains a list summarizing the data file commands in SmartBASIC version 1.0.

Most operating system commands will accept a volume number (V#) and a slot number (S#) specification. In the minimal configuration of ADAM, these have no effect and are not mentioned in the following list. You can tell ADAM which drive contains a file using the D# specification. If a drive is not specified, ADAM will look only on the default drive. The default drive (D1) is the left digital data pack, except when SmartBASIC

has been loaded from a disk drive. In that case the default drive is D3. You should also note that the operating system (OS) distinguishes between upper- and lowercase letters in file names (HELLO and hello are different file names).

The form of the commands, as shown in the list, is the form used in direct mode, although many of the commands don't work in direct mode. Here, as in the table, *filename* must be replaced by the name of the file you wish to use.

The form of each command, in programming mode, is:

```
PRINT CHR$(4);"formatted form of command"
```

The string shown in quotation marks may be replaced by the contents of a string variable.

Data File Handling Commands

APPEND — adds data to the end of an already existing sequential file. No WRITE command is needed after an APPEND.

format: APPEND *filename*

(Note: the file must be closed after an APPEND.)

BLOAD — loads a binary file into memory at the location specified.

format: BLOAD *filename*, A# [,D#] (where A# is the memory address and D# is the optional drive number).

BRUN — runs a binary program by jumping to the starting address automatically.

format: BRUN *filename* A# [,D#] (The A# and D# have preset default values that can be changed. See BLOAD for A# and D#.)

BSAVE	saves binary information on the digital data pack. format: BSAVE *filename*, A#,L# [,D#] (The A# and L# are *not* optional and refer to the starting address and the length, respectively, of the binary information to be saved.)
CATALOG	provides a list of all files on the specified data pack. format: CATALOG [,D#]
CLOSE	closes the specified file on the data pack. format: CLOSE *filename* [,D#]
DELETE	erases the designated file from the specified data pack. format: DELETE *filename* [,D#]
INIT	wipes out the directory on a data pack. format: INIT *volumename* [,D#] (volumename is *not* optional.)
LOAD	loads program with designated filename from data pack. format: LOAD *filename* [,D#]
LOCK	protects a data pack file from deletion until unlocked. format: LOCK *filename* [,D#]
OPEN	prepares a data pack file for access. It sets up a buffer in memory and adds the filename to the directory on the data pack (if necessary). For random access files the length parameter L# must be specified. In the example, L200 refers to the length of each record (or entry) in the file.

format: OPEN *filename* [,L#] [,D#]

example (random access file): PRINT CHR$(4); "OPEN NAMES, L200"

POSITION — moves the data pack pointer R# records (or entries) forward from its current position. This command is used only with ASCII or sequential files, *not* with random access files.

format: POSITION *filename* [,R#]

READ — designates a data pack file from which subsequent INPUT and GET commands receive data. The file must be opened before being read. The file is read starting at the first entry. For random access files, the record number parameter is required. Also if a byte number parameter is used, all INPUT and GET commands obtain data starting at the specified byte.

format: READ *filename* [,R#] [,B#]

RECOVER — allows access to a backup file (a) after renaming of a primary file (A).

format: *RECOVER* filename

RENAME — changes the name of a data pack file without changing its contents.

format: RENAME $filename_1$,$filename_2$ [,D#]

RUN — loads and executes a program from data pack storage.

format: RUN *filename* [,D#]

SAVE — permanently stores program on the data pack with the specified filename.

format: SAVE *filename* [,D#]

UNLOCK — cancels the locked status of a data pack file. It may then be changed or deleted.

format: UNLOCK *filename* [,D#]

WRITE — designates the data pack file to which subsequent PRINT statements are directed. Positions the file pointer at the first entry in the file.

format: WRITE *filename* [,R#] [,B#]

Note: In programming mode, all commands shown in the table must be in quotes and be preceded by the command to direct information to the operating system. This command is:

```
PRINT CHR$(4);
```

Remember that the cursor must be in the left most column before this command is issued.

In programming mode, any operating system command will cancel all previous commands. Particularly useful is the null command:

```
PRINT CHR$(4)
```

Summary

The last section is a summary of all the file handling commands in SmartBASIC. Table 16.2 contains some other operating system commands and table 16.3 contains the various forms of the MON command.

BUZZwords

ACCESS

BUFFER

FILE POINTER

FILES

HARDWARE FAILURE

I/O ERROR

MONITOR

OPERATING SYSTEM

READ FILE

RESERVED WORDS

WRITE FILE

AFTERWORD Playing in ADAM's Brain

The commands needed to fool around in ADAM's brain are:

PEEK POKE CALL LOMEM HIMEM

These commands are, at present, the only way to look into ADAM's memory, create new shapes, make noise and — even music.

We saved the discussion of talking to ADAM *without* translation into machine language for this Afterword. The subject of this book is BASIC, not machine language programming. However, there are a few fun uses of machine language and we didn't want to write a book about ADAM without showing them to you! So we decided to have an Afterword with short de-

scriptions of the SmartBASIC commands that interact with ADAM without translation. We wrote four programs using a mixture of previous SmartBASIC commands and the new ones. The two short programs are a memory peeker and noise generator. They're short enough so that we'll try to explain them. The other two are very long and we won't give any technical description of how they work - we only tell you how to use them. Thus far we've tried to explain almost every program. We can't do that with the Smart ShapeMaker and Smart Musician without writing another book.

Many programs are worth using without trying to write your own versions of them. For example, we don't usually think it worthwhile writing a 1000 (or more) line program to do something if we can buy a program for twenty dollars that does it. A long program can take weeks (or months) to write, enter and debug. (We might only write a program like an existing one when we're curious about how it works.) Do what interests you - you'll learn more and be happier doing it.

By now you're close to mastering the Coleco version of BASIC — SmartBASIC. SmartBASIC is only one of the languages that can communicate with ADAM. Others, such as LOGO, are either available or soon will be available. SmartBASIC is an interpreter - each statement is translated one at a time into ADAM's internal language. After each translation is done ADAM will process that statement. However, even outside of computers an interpretation isn't always as effective as we would like. For example, interpreted conversations must be slower than those conducted in one language. We talked about the need for speed in Chapter 15 when we wanted to animate hi-res figures. ADAM is incredibly fast. Except for the way ADAM handles files, ADAM's BASIC is faster than the one supplied with computers costing four times as much. Yet even with SmartBASIC's speed there are times when we need still more speed. There may also be words in one language that don't exist or aren't as clear in another. ADAM, like all computers, has the same problems with translations. In our situation the translation is going from SmartBASIC into ADAM's machine language. Some of what can't be asked in (the current version of) SmartBASIC include:

1. To make new shapes (see Chapter 15 for the built-in shape.)
2. To use ADAM's marvelous sound capabilities
3. To chain programs (see Chapter 16)
4. To change the way that the printer and display screen work.

1. How SmartBASIC works

When we first got our ADAMs, there wasn't much written about SmartBASIC. Obviously we couldn't write a book like this without knowing all the commands. We decided the easiest way to find them was to look in ADAM's memory after we loaded SmartBASIC. Fortunately, most BASICs have commands that let you look at the contents of individual memory locations, change those contents and even run programs written in machine language.

The command for looking into a particular memory location is display PEEK(expernm). *exprnm*, as we've already mentioned, is the abbreviation for a number or something that ADAM can evaluate as a number. The *exprnm* must be between 0 and 65536. If

$$a = \text{PEEK}(2001)$$

and we tell ADAM to

PRINT *a*

the screen will show the contents of memory location 2001. (For experts, *exprnm* is in decimal, not hex. You can also use negative numbers. These count down from 65535.) *a* will always be some number between 0 and 255. However, location 2001 may actually hold a letter. (ADAM sets up a table that tells it whether location 2001 holds a character, a number, some part of SmartBASIC or a part of the operating system.) To print the contents of location 2001 including a possible character, we used

PRINT CHR$(a)

While we were PEEKing to find the commands we assumed that the letters for each command word were next to each other. So we printed out, in order, the character equivalent of the contents of every memory location available to SmartBASIC. This procedure was more successful than we had expected. We found all the commands, all of ADAM's reserved words, the copyright notice from the name of the company that wrote SmartBASIC for Coleco (and a message from the programmers who wrote SmartBASIC to someone named Cathy). The program to PEEK in ADAM's memory is simple. Before using it turn ADAM off and on again and then load SmartBASIC. Otherwise you risk picking up lots of junk left

over from earlier programs. Remember, if some byte can be misinterpreted, it *will* be.

```
10 REM probing ADAM's memory
20 PRINT "Get 4 sheets of paper ready"
30 PRINT "Press any key when ready"
40 GET a$
50 PR#1
60 FOR i = 0 TO 65535
70 a% = PEEK(i)
80 IF a% < 91 AND a% > 64 THEN PRINT CHR$(a%);
90 IF a% < 123 AND a% > 96 THEN PRINT CHR$(a%);
100 NEXT i
110 PR#0
120 END
```

(Remember the CONTROL/S combination stops the printout. This lets you change paper. Then press any key to resume printing.)

ADAM also has a command to place a number into a memory location. The command is POKE. It's format is:

POKE *location, exprnm*

The location is the memory location that you want to place *exprnm* into. *exprnm* must be between 0 and 255. The command is *too* easy to use. It's easy to write a program that POKEs a random integer between 0 and 255 into a random memory location. **Don't do it**. ADAM will *crash* with just a few misplaced POKEs. If you play with POKEs, and get into trouble, it's easy to get out of: either reload SmartBASIC or turn ADAM off for a few minutes and restart. (It's impossible to be sure, but we find it hard to believe you can harm ADAM with POKEs by, for example, telling the data drives to turn in opposite directions at the same time.)

POKEs should be used for several purposes:

1. To place data in specific memory locations - for example, shape table data.
2. To place machine language programs or subroutines into memory - for example, a subroutine to send data to the sound generator.
3. To make direct use of some of the machine language subroutines built into SmartBASIC.

The programs in the Afterword do the first two. Coleco hasn't released enough technical information about SmartBASIC at this time for the third.

2. The Smart ShapeMaker

In Chapter 15 we showed you how to use ADAM's built in shape. A square with a line through it isn't the most exciting of figures - even animated. The program in this section lets you use the arrow keys to create a shape. After the shape is stored, the program can use its built in shape-manipulator. You can also END the program and then write your own program to manipulate the shapes. If you do this then the very first statement in your program should be:

```
HIMEM: 45000
```

HIMEM blocks off a portion of ADAM's memory. The ShapeMaker stores the shapes in ADAM's memory starting at register 45000. By using the HIMEM command to place a barrier here, you prevent the (unlikely though it is) possibility that any program you write after ENDing the ShapeMaker will over-write the information describing the shape. We always play it safe and use this HIMEM command.

Trick of the Trade One warning about the HIMEM command. Experimenting with the HIMEM command is dangerous - unless you know a bit about ADAM's memory map. If you set it too high your programs will crash in a big way. (This can also happen if you POKE around 53000). Loading or writing a new program, as well as entering the command NEW, resets HIMEM to its original location.

Some hints on entering this program: it's long - so be careful. One idea is to enter it module by module. (They're set off by rows of *'s). We strongly suggest taking frequent breaks from typing and if possible asking someone else to help find typos. After each section is entered use the Printer to print it out. Then compare each line with the printed version. You can use the word processing program to enter it, but then be careful that each line contains no syntax errors. Otherwise, as we said in Interlude 1, when you load it in SmartBASIC the line will disappear. In any case, save the partially entered version every few minutes. After the program has been *completely* entered and debugged you might want to save it on a freshly INITed data

pack. A fresh data pack is best for storing very long programs - ADAM loads long programs more quickly from a (relatively) empty data pack.

Before using the ShapeMaker we suggest a quick glance at Section 1 of Chapter 15. The program isn't terribly useful if you haven't mastered how to use shapes.

The ShapeMaker is pretty much self-explanatory when it's RUNning. You may discover it's a good idea to make a rough sketch of your shape before using the program to store it in ADAM's memory. Finally, the only cautions are that each time you use it:

1. Any shapes in memory are lost - (Save the ones you like on a data pack!)
2. DON'T try to retrieve a non-existent SHAPE file. The program may crash spectacularly. If it does you'll have to reload SmartBASIC.

Here's the program

```
 10 REM      a shape table generator
 20 HIMEM :45055
 30 DIM bt%(5000), smnp%(40, 5)
 40 nmsh% = 1: pvshp = 45097
 50 DIM vt(1): REM      each byte has two halves
 60 pnup% = 1: idex% = 0
 70 op$ = "open ": cl$ = "close "
 80 wr$ = "write ": rd$ = "read "
 90 d$ = CHR$(4)
100 REM *********END INITIAL ASSIGNMENTS********
150 HOME: VTAB (8)
160 INVERSE
170 PRINT "       The SmartShape Maker      ": NORMAL
180 PRINT: PRINT "You can make up to 20 shapes"
190 PRINT "Press any key to start"
200 GET qw$
210 REM *********SHAPE TABLE INITIALIZATIONS****
260 POKE 16766, 0: REM      set up shape table
270 POKE 16767, 176: REM      shapes start at 45098
280 POKE 45056, 20: REM      number of shapes
290 POKE 45057, 0
499 REM *********MAIN MENU**********************
500 TEXT
510 VTAB (8)
520 PRINT "1. Create New SHAPE"
530 PRINT "2. Manipulate SHAPES in memory"
540 PRINT "3. Retrieve SHAPEs from Data       Pack"
```

(continued)

```
  550 PRINT "4: END"
  560 PRINT: PRINT
  570 GOSUB 4900
  580 GET ch$: ch = VAL(ch$)
  590 ON ch GOTO 1000, 2000, 4000, 700
  600 HOME: GOSUB 4600
  610 PRINT: PRINT: PRINT
  620 GOTO 510
  699 REM *********EXIT and CLEAR SCREEN**********
  700 TEXT
  710 END
  999 REM     ********DIRECTIONS for DRAWING*******
 1000 IF nmsh% <= 20 THEN 1070
 1020 PRINT "TOO many shapes - over 20 "
 1030 PRINT "Returning to menu"
 1040 tim = 2: GOSUB 4500
 1050 GOTO 500
 1070 HOME
 1080 VTAB (3)
 1090 PRINT "Use the arrow keys to make the shape."
 1100 PRINT "Pressing the p changes the pen"
 1110 PRINT "position": PRINT
 1120 PRINT "When the PEN is down ADAM will"
 1130 PRINT "plot a dot.  When UP it will "
 1140 PRINT "move without plotting.": PRINT
 1150 PRINT "Press e to END.": PRINT
 1160 PRINT: PRINT "You'll be making shape number  "
 1170 INVERSE: PRINT: PRINT: HTAB (15): PRINT " "; nm
shp%; " "
 1180 NORMAL: PRINT: PRINT
 1200 PRINT "Press any key to start"
 1210 GET a$
 1220 lum = 128: rw = 96: idex% = 0
 1230 HGR: HCOLOR  = 3
 1240 pnup% = 1
 1250 GOSUB 4700
 1270 REM  *******BEGIN drawing module*************
 1280 idex% = idex%+1: REM      counts number of bytes
 in shape
 1290 vt(0) = 0: vt(1) = 0
 1300 FOR ha = 0 TO 1: REM      each half of byte
 1310 GET a$: a = ASC(a$)
 1320 IF a$ = "p" OR a$ = "P" THEN  pnup% =  NOT (pnup%):
GOSUB 4700: GOTO 1310
 1330 IF a$ = "e" OR a$ = "E" THEN  GOSUB 1500: GOTO
 1310
 1340 IF a = 160 THEN  vt(ha) = 0: rw = rw-1
 1350 IF a = 161 THEN  vt(ha) = 1: lum = lum+1
 1360 IF a = 162 THEN  vt(ha) = 2: rw = rw+1
 1370 IF a = 163 THEN  vt(ha) = 3: lum = lum-1
 1380 IF pnup% = 1 THEN  HPLOT lum, rw
 1390 NEXT
```

(continued)

```
 1420 bt% = (4*pnup%+vt(0))+(4*pnup%+vt(1))*8
 1421 REM ****(convert to hexadecimal code)****
 1430 IF bt% <> 0 THEN  bt%(idex%) = bt%
 1440 IF bt% = 0 THEN  bt%(idex%) = 8: idex% = idex%+
1: bt%(idex%) = 24
 1441 REM  ***(rework two moves up no plotting*****
 1450 GOTO 1280
 1500 HOME
 1510 PRINT "Are you sure? Press r to"
 1520 PRINT "continue DRAWING - anything"
 1530 PRINT "else to go to STOREing menu";
 1540 GET yn$
 1550 IF yn$ = "r" OR yn$ = "R" THEN  GOSUB 4700: RETURN
 1600 REM ********mark end of shape************
 1610 bt% = (4*pnup%+vt(0))
 1620 IF bt% = 0 THEN  bt%(idex%) = 0: GOTO 1650
 1630 IF bt% <> 0 THEN  bt%(idex%) = bt%: idex% = idex%+1
 1640 bt%(idex%) = 0
 1650 POP: GOTO 3000
 1999 REM  ****START SHAPE MANIPULATOR*************
 2000 TEXT: VTAB (5)
 2020 PRINT "    ShapeManipulator": PRINT
 2030 PRINT "You have stored "; nmsh%-1; " shapes"
 2040 PRINT: PRINT
 2050 PRINT "Wait a moment": PRINT
 2100 FOR i2 = 1 TO 40
 2110 FOR j2 = 1 TO 5
 2120 smnp%(i2, j2) = 0
 2130 NEXT: NEXT
 2140 sn = 1
 2150 PRINT "After the shapes appear, press any key
to go back to menu."
 2160 PRINT: PRINT "To stop enter a negative number":
 PRINT
 2170 PRINT "What shape number?": PRINT
 2180 INPUT smnp%(sn, 0)
 2190 IF smnp%(sn, 0) <= 0 THEN 2320
 2200 IF smnp%(sn, 0) > nmsh% THEN  GOSUB 4600: GOTO
2170
 2210 INPUT "What HCOLOR CODE?"; smnp%(sn, 1)
 2220 IF smnp%(sn, 1) < 0 OR smnp%(sn, 1) > 15 THEN
GOSUB 4600: GOTO 2210
 2230 INPUT "What SCALE? "; smnp%(sn, 2)
 2240 IF smnp%(sn, 2) < 1 OR smnp%(sn, 2) > 255 THEN
 GOSUB 4600: GOTO 2230
 2250 INPUT "What ROTation?"; smnp%(sn, 3)
 2260 IF smnp%(sn, 3) < 0 OR smnp%(sn, 3) > 64 THEN
GOSUB 4600: GOTO 2250
 2270 INPUT "What column?"; smnp%(sn, 4)
 2280 IF smnp%(sn, 4) < 0 OR smnp%(sn, 4) > 255 THEN
 GOSUB 4600: GOTO 2270
 2290 INPUT "What row?"; smnp%(sn, 5)
```

(continued)

```
2300 IF smnp%(sn, 5) < 0 OR smnp%(sn, 5) > 191 THEN
GOSUB 4600: GOTO 2290
2310 sn = sn+1: HOME: GOTO 2160
2320 HGR2
2330 FOR i3 = 1 TO sn
2340 IF smnp%(i3, 0) < 0 THEN 2450
2360 HCOLOR  = smnp%(i3, 1)
2370 SCALE  = smnp%(i3, 2)
2380 ROT  = smnp%(i3, 3)
2390 DRAW smnp%(i3, 0) AT smnp%(i3, 4), smnp%(i3, 5)
2400 NEXT
2450 GET e$: TEXT
2460 GOTO 500
2499 REM ********CLEAR SCREEN and EXIT*********
3000 TEXT: VTAB (8)
3010 PRINT "1. Store last shape in memory": PRINT
3020 PRINT "2. Store last shape on data   ";
3030 PRINT "    pack": PRINT
3040 PRINT "3. Store last shape on BOTH": PRINT
3050 PRINT "4. DO NOT SAVE SHAPE in any way"
3060 GOSUB 4900
3070 GET ch$: ch = VAL(ch$)
3110 IF ch = 1 THEN  GOSUB 3200: GOTO 500
3120 IF ch = 2 THEN  GOSUB 3400: GOTO 500
3130 IF ch = 3 THEN  GOSUB 3400: GOSUB 3200: GOTO 500
3140 IF ch = 4 THEN 500
3150 HOME: GOSUB 4600
3160 GOTO 3010
3199 REM  ****STORE CURRENT SHAPE IN MEMORY********
3200 FOR q2 = 1 TO idex%
3210 POKE pvsh+q2, bt%(q2)
3230 NEXT
3240 bnsh = pvsh-45055
3250 ho = INT(bnsh/256)
3260 lo = bnsh-(ho*256)
3270 POKE 45056+2*nmsh%, lo
3280 POKE 45057+2*nmsh%, ho
3320 pvsh = pvsh+idex%
3330 nmsh% = nmsh%+1
3340 RETURN
3399 REM  *********STORE SHAPE ON DATA PACK********
3400 ONERR  GOTO 4800
3410 TEXT: VTAB (12)
3420 GOSUB 5000
3430 PRINT "Enter name of shape to be      ";
3440 PRINT "stored"
3450 INPUT sn$
3460 PRINT: PRINT
3470 PRINT "Wait a moment please."
3490 PRINT d$; op$+sn$
3500 PRINT d$; wr$+sn$
3510 FOR q2 = 1 TO idex%
```

(continued)

```
3520 PRINT bt%(q2)
3530 PRINT d$; op$+sn$
3540 PRINT d$; wr$+sn$
3550 FOR q2 = 1 TO idex%
3560 PRINT bt%(q2)
3570 NEXT
3580 PRINT d$; cl$+sn$
3590 CLRERR
3600 RETURN
3999 REM   *********RETRIEVE SHAPE FROM DATA PACK*
4000 ONERR  GOTO 4800
4010 TEXT: VTAB (12)
4020 GOSUB 5000
4030 PRINT "Name of shape to be retrieved"
4040 PRINT: PRINT
4050 INPUT sn$
4060 HOME: VTAB (5)
4070 PRINT "Wait a moment please"
4080 i5 = 1
4090 PRINT d$; op$+sn$
4100 PRINT d$; rd$+sn$
4120 INPUT a5
4130 bt%(i5) = a5
4140 IF bt%(i5) = 0 THEN  HTAB (1): GOTO 4160
4150 i5 = i5+1: GOTO 4120
4160 idex% = i5
4170 PRINT d$; cl$+sn$
4180 CLRERR
4190 GOSUB 3200
4200 GOTO 500
4499 REM *********TIMING LOOP*****************
4500 FOR q = 1 TO 1000*tim
4510 NEXT q
4520 RETURN
4600 PRINT "You entered something strange"
4610 PRINT "Please retry"
4620 PRINT
4630 RETURN
4699 REM      *********POSITION OF PEN IN DRAWING*
4700 HOME
4710 IF pnup% = 0 THEN  PRINT "pen is UP"
4720 IF pnup% = 1 THEN  PRINT "pen is DOWN"
4730 RETURN
4799 REM  *********begin messages and error traps*
4800 TEXT: VTAB (12)
4810 IF ERRNUM(1) = 7 THEN  PRINT "File not found -Try
a different   data pack"
4820 IF ERRNUM(1) = 8 THEN  PRINT "Problem with data
pack"
4830 IF ERRNUM(1) = 9 THEN  PRINT "No more room-try
a different data pack"
4840 PRINT "If the error has been corected"
```

(continued)

```
4850 PRINT "press r.": PRINT: PRINT
4860 PRINT "Press anything else if not."
4870 GET e2$
4880 IF e2$ = "r" OR e2$ = "R" THEN  RESUME
4890 STOP
4900 PRINT: PRINT
4910 PRINT "Please press the number of your";
4920 PRINT "choice.": RETURN
5000 PRINT: PRINT: PRINT
5010 PRINT "Do you want to see the CATALOG"
5020 PRINT "Press y for yes - press "
5030 PRINT "anything else if not"
5040 GET yn$
5050 IF yn$ <> "y" AND yn$ <> "Y" THEN 5090
5060 HOME
5070 PRINT: PRINT "Wait a moment please"
5080 PRINT d$; "CATALOG"
5090 RETURN
```

3. ADAM makes noise

You've already seen that ADAM can draw beautiful pictures. If you've played any game, you know that ADAM can also create sound - play melodies and make noises like gunfire. ADAM has a built in chip called the SN76489A, made by the Texas Instrument Company, that can play music using three voices or make eight types of noise. This is the same chip used in many more expensive machines. The voices work like different people singing together - they can each sing separate notes. We'll leave music to the next section and discuss noise here, although there is a lot of overlap in how they work.

In order to produce sound, the sound chip must be told what to do. Unfortunately there are no SmartBASIC instructions (yet) to do the telling! So the instructions have to be written in machine language. It's easy to describe the machine language program in words. The sound chip sits outside port 255 (see Chapter 2 for a discussion of ports). Our machine language subroutine, *sender*, looks in a specific memory location for the data to send to the sound chip. Then it kicks that information out port 255. The last thing it does is execute a return back into a program written in SmartBASIC. In the noisemaker subroutine, program on page 458, lines 1000-1030 place *sender* into memory. *Sender* looks at *place%* (memory location 27456). The CALL command is like a GOSUB, only CALL is used with machine language subroutines. Instead of using a line number (like a subroutine), the location in memory is used. In the noise maker program, the command is:

```
CALL send%
```

Since *send%* is 27450, the command is the same as:

CALL 27450

To summarize: the subroutine first POKEs data into *place%* and then *sender* kicks it out port 255.

Here's the subroutine:

```
495 REM
496 REM
497 REM  ************************************************
498 REM     the noise sender
499 REM  ************************************************
500 n1% = 224 + type%
510 n2% = 255 - loud%
520 n3% = 255
530 FOR repeat = 1 TO repeat%
540 POKE place%,n1%: CALL send%
550 POKE place%,n2%: CALL send%
560 FOR i = 1 TO 1000*dur: NEXT i
570 POKE place%,n3%: CALL send%
580 FOR i = 1 TO 1000*pause: NEXT i
590 NEXT repeat
600 RETURN
1000 READ send%
1010 FOR memloc = send% TO send% + 5
1020 READ entry: POKE memloc,entry
1030 NEXT memloc
1040 place% = send% + 6
1050 RETURN
10000 DATA 27450,58,64,107,211,255,201
```

You get a different kind of noise by replacing the 224 in line 500 by 228. Experiment to find what you like.

The subroutine must be fed several different pieces of information. First the noise volume, *loud%*, is a number between 0 and 15. 0 is *off* and 15 is the loudest. The noise type, *type%*, is either 0, 1 or 2. The duration, *dur*, can be any positive number. We suggest starting with *dur* = .1; *pause* is the pause between noises, it can be any positive number - try .5 to start off. *Repeat* is the number of times you want the noise to repeat.

One way to experiment with ADAM's noise generating capabilities is to attach the following driver program that calls the subroutine. (A *driver* is a program, subroutine or piece of machinery that tells something else to do work.)

```
10 REM  making ADAM noisy
20 LOMEM :27500
30 GOSUB 1000: REM set up sender
40 INPUT "noise volume (0-15)";loud%
50 INPUT "noise type (0,1 or 2)";type%
60 INPUT "duration";dur
70 INPUT "pause between";pause
80 INPUT "repeat";repeat%
90 GOSUB 500
100 END
```

We used a new command here. LOMEM puts a barrier at some memory location. ADAM will not put a SmartBASIC program, data or variables below it - in this case below 27500. It's similar to the command HIMEM from the previous section. Both HIMEM and LOMEM keep SmartBASIC from using parts of memory that contain (other) important things. The default values of HIMEM and LOMEM keep programs away from the operating system and the SmartBASIC interpreter. When using machine language programs or data (like shape tables), change HIMEM or LOMEM to make sure that ADAM doesn't overwrite your data. (In future versions of SmartBASIC, the default values of HIMEM and LOMEM may be changed. If so Coleco is sure to have the new values.)

4. The Smart Musician

The program in this section is, by far, the longest in the book. Using it you can write and play music with all of ADAM's three voices. We could have written a much shorter program but then you would have to do all the translations that ADAM can easily do. We also wanted to make the program as friendly as possible; so we made the whole program menu driven. We'll first give you hints on entering it, then we'll give directions on how to use it.

Entering the Smart Musician program

We can't imagine anyone who could enter the program on page 460 into ADAM without making a few mistakes. In a program as long as this a few typos are inevitable - and incredibly hard to find. To minimize problems, here are a few tricks to try. First, line 20 is missing. This isn't a mistake. Line 20 will contain an ONERR command - to be inserted later. An ONERR command prevents ADAM from showing you error messages. When debugging a program you want all the help you can get. Leave off any error traps until the rest of the program (except for the file-handling modules) is correct. Then enter the following error trap:

```
 190 VTAB 21: HTAB 2: PRINT "Press the SmartKey with"
 200 HTAB 5: PRINT "that number to start";
 210 GET ch$
 220 HOME
 230 ch% = ASC(ch$)-128
 240 IF ch% < 1 THEN  GOSUB 300: GOTO 110
 250 ON ch% GOSUB 7000, 1500, 1700, 5000, 270
 260 GOTO 110
 270 END
 300 HOME: FLASH: VTAB 12: HTAB 4: PRINT "Your choice is
 not allowable": NORMAL
 310 FOR i = 1 TO 50: PRINT CHR$(7); : NEXT i
 320 RETURN
 998 REM  set up machine language program
1000 READ send%
1010 FOR memloc = send% TO send%+5
1020 READ entry: POKE memloc, entry
1030 NEXT memloc
1040 place% = 27456
1050 RETURN
1198 REM  set up note decoder
1200 mm$ = "CDEFGAB"
1210 FOR i = 1 TO 12: READ mm(i): NEXT
1220 RETURN
1498 REM  play an untranslated selection
1500 k9 = 0: GOSUB 4700
1510 HTAB (1): PRINT d$+op$+phyle$
1520 PRINT d$+rd$+phyle$
1530 t = 0
1540 t = t+1: INPUT note$(t)
1550 IF note$(t) = "z" THEN 1570
1560 GOTO 1540
1570 HTAB (1): PRINT d$+cl$+phyle$
1580 sound%(t, 0) = -1
1590 tt = t-1
1600 FOR t = 1 TO tt: a$ = note$(t): GOSUB 2000: NEXT t
1610 GOSUB 5000
1620 GOSUB 3500
1630 RETURN
1698 REM  play a translated selection
1700 k9 = 0: GOSUB 4700
1710 PRINT: PRINT d$+op$+phyle$
1720 PRINT d$+rd$+phyle$
1730 t = 0
1740 t = t+1
1750 FOR i = 0 TO 4
1760 INPUT sound%(t, i)
1770 IF sound%(t, i) = -1 THEN 1800
1780 NEXT i
1790 GOTO 1740
1800 PRINT: PRINT d$+cl$+phyle$
1810 GOSUB 5000
```

(continued)

```
1820 RETURN
1998 REM          note decoder
2000 len = LEN(a$)
2010 oct% = VAL(MID$(a$, 2, 1))
2020 GOSUB 2100: REM  find frequency
2030 GOSUB 2300: REM  find duration
2040 GOSUB 2500: REM  find the loudness
2050 GOSUB 2700: REM  find the voice
2060 GOSUB 2800: REM  enter sound%(t,.)
2070 RETURN
2098 REM   find the frequency
2100 FOR i = 1 TO 7
2110 IF MID$(a$, 4, 1) = MID$(mm$, i, 1) THEN  j9 = 5:
GOTO 2160
2120 NEXT i
2130 j9 = 5
2140 IF len = 4 THEN  freq = mm(i)*2^(oct%-1): RETURN
2150 j9 = 5
2160 IF MID$(a$, 5, 1) = "#" AND i < 3 THEN  i = i+7: j
9 = 6
2170 IF MID$(a$, 5, 1) = "#" AND i > 3 AND i < 8 THEN
i = i+6: j9 = 6
2180 freq = mm(i)*2^(oct%-1)
2190 RETURN
2298 REM  duration 1
2300 IF len = 4 THEN  RETURN
2310 FOR j = j9 TO len
2320 IF MID$(a$, j, 1) = "d" THEN  GOSUB 2400
2330 NEXT j
2340 RETURN
2398 REM  duration 2
2400 IF len = j+1 THEN  dur% = VAL(MID$(a$, len, 1))
2410 IF len = j+2 THEN  dur% = VAL(MID$(a$, len-1, 2))
2420 m9$ = MID$(a$, j+2, 1)
2430 IF m9$ = "L" OR m9$ = "V" THEN  dur% = VAL(MID$(a$
, j+1, 1)): j9 = j+2: POP: RETURN
2440 dur% = VAL(MID$(a$, j+1, 2)): j9 = j9+3: RETURN
2498 REM  loudness 1
2500 IF j9 > len THEN  RETURN
2510 FOR j = j9 TO len-1
2520 IF MID$(a$, j, 1) = "L" THEN  GOSUB 2600
2530 NEXT j
2540 RETURN
2598 REM  loudness 2
2600 IF j+1 = len THEN  loud% = VAL(MID$(a$, len, 1)):
POP: RETURN
2610 IF MID$(a$, j+2, 1) = "V" THEN  loud% = VAL(MID$(
a$, j+1, 1)): j9 = j9+2: POP: RETURN
2620 loud% = VAL(MID$(a$, j+1, 2)): j9 = j9+3: POP: RETURN
2698 REM  loudness 2
2700 IF j9 > len THEN  voice% = 1: RETURN
2710 voice% = VAL(MID$(a$, len, 1)): RETURN
```

(continued)

```
 2798 REM  assemble translated note
 2800 freq% = 3597000/(32*freq)
 2810 sound%(t, 4) = voice%
 2820 sound%(t, 0) = 112+sound%(t, 4)*32+16-loud%
 2830 sound%(t, 3) = dur%
 2840 sound%(t, 1) = freq%/16
 2850 sound%(t, 2) = 128+(freq%-16*sound%(t, 1))+(voice%
-1)*32
 2860 RETURN
 2998 REM  translator
 3000 k9 = 0: GOSUB 4700
 3010 t = 0
 3020 PRINT d$+op$+phyle$
 3030 PRINT d$+rd$+phyle$
 3040 t = t+1: INPUT a$: note$(t) = a$
 3050 IF a$ <> "z" THEN  GOSUB 2000: GOTO 3040
 3060 t = t+1: sound%(t, 0) = -1
 3070 GOSUB 5000
 3080 GOSUB 5000
 3090 GOSUB 3500
 3100 RETURN
 3498 REM  save a translated version
 3500 HOME: VTAB 10: HTAB 3: PRINT "Do you want to save"
 3510 HTAB 3: PRINT "the translated version?"
 3520 HTAB 6: PRINT "(answer y or n)"
 3530 GET a$
 3540 IF a$ = "Y" OR a$ = "y" THEN 3560
 3550 RETURN
 3560 k9 = 1: GOSUB 4700
 3570 GOSUB 3700
 3580 RETURN
 3698 REM  save a translated version 2
 3700 PRINT: PRINT d$+op$+phyle$
 3710 PRINT d$+wr$+phyle$
 3720 t = 0
 3730 t = t+1
 3740 FOR i = 0 TO 4
 3750 PRINT sound%(t, i)
 3760 IF sound%(t, i) = -1 THEN 3790
 3770 NEXT i
 3780 GOTO 3730
 3790 PRINT d$+cl$+phyle$
 3800 GOSUB 3900
 3810 RETURN
 3898 REM  display selection filed
 3900 HOME: INVERSE: VTAB 10: HTAB 3
 3910 PRINT "The selection has been filed": NORMAL
 3920 FOR i = 1 TO 10000: NEXT i
 3930 RETURN
 4698 REM  file prompt display
 4700 HOME
 4710 VTAB 5: HTAB 3: PRINT "If you would like"
```

(continued)

```
 4720 HTAB 3: PRINT "to see the catalog, press y"
 4730 HTAB 3: PRINT "Otherwise, press another key"
 4740 GET yn$
 4750 IF yn$ = "Y" OR yn$ = "y" THEN  HTAB 1: PRINT d$;
"catalog"
 4760 HTAB 2: PRINT "Type the name of the file"
 4770 IF k9 = 0 THEN  HTAB 5: PRINT "containing the music,"
 4780 IF k9 = 1 THEN  HTAB 5: PRINT "to hold the music,"
 4790 INPUT phyle$
 4800 HOME: VTAB 10: HTAB 8: PRINT "Wait a moment"
 4810 RETURN
 4998 REM  the Smart Musician
 5000 HOME: VTAB 8: HTAB 3: PRINT "How many times do you
want"
 5010 HTAB 3: PRINT "to repeat the selection?"
 5020 HTAB 3: INPUT repeat%
 5030 VTAB 15: HTAB 3: PRINT "Select some integer"
 5040 HTAB 3: PRINT "(whole number) as tempo"
 5050 HTAB 3: PRINT "Then press return"
 5060 HTAB 3: PRINT "Try 20 as a first value"
 5070 HTAB 3: INPUT tempo%
 5080 GOSUB 5500: REM  play the music
 5090 RETURN
 5498 REM  play music
 5500 FOR repeat = 1 TO repeat%
 5510 n0% = 0
 5520 n0% = n0%+1: n1% = n0%
 5530 IF sound%(n0%, 0) = -1 THEN 5610
 5540 IF sound%(n0%, 4) <> 1 THEN  n0% = n0%+1: GOTO 5530
 5550 FOR i = n1% TO n0%
 5560 PRINT i: GOSUB 5800: REM  turn on sound
 5570 NEXT
 5580 GOSUB 5900: REM  timer
 5590 GOSUB 6000: REM  turn off sound
 5600 GOTO 5520
 5610 NEXT repeat
 5620 RETURN
 5798 REM  turn on sound
 5800 FOR ii = 2 TO 0 STEP -1
 5810 POKE place%, sound%(i, ii): CALL send%
 5820 NEXT
 5830 RETURN
 5898 REM  timer
 5900 FOR ii = 1 TO tempo%*sound%(n1%, 3): NEXT
 5910 RETURN
 5998 REM  turn off sound
 6000 FOR ii = 1 TO 3
 6010 POKE place%, 127+32*ii: CALL send%
 6020 NEXT
 6030 RETURN
 6998 REM  main editing menu
 7000 HOME: VTAB 3: HTAB 6: PRINT "MAIN EDITING MENU"
```

(continued)

```
 7010 VTAB 6: HTAB 3: PRINT "I. Enter"
 7020 VTAB 8: HTAB 3: PRINT "II. Insert"
 7030 VTAB 10: HTAB 3: PRINT "III. Delete"
 7040 VTAB 12: HTAB 3: PRINT "IV. Play"
 7050 VTAB 14: HTAB 3: PRINT "V. Save untranslated piece"
 7060 VTAB 16: HTAB 3: PRINT "VI. Back to main menu"
 7070 VTAB 19: HTAB 2: PRINT "Read the directions in"
 7080 HTAB 2: PRINT "book before proceeding"
 7090 VTAB 22: HTAB 2: PRINT "Press the SmartKey with"
 7100 HTAB 5: PRINT "the number of your choice";
 7110 GET ch$
 7120 HOME
 7130 ch% = ASC(ch$)-128
 7140 IF ch% < 1 THEN  GOSUB 300: GOTO 7000
 7150 ON ch% GOSUB 8700, 8300, 7500, 7400, 7200, 7300
] 7160 GOTO 7000
 7198 REM  save untranslated piece
 7200 k9 = 1: GOSUB 4700
 7210 i = 0
 7220 PRINT: PRINT d$+op$+phyle$
 7230 PRINT d$+wr$+phyle$
 7240 i = i+1: PRINT note$(i)
 7250 IF note$(i) = "Z" OR note$(i) = "z" THEN 7270
 7260 GOTO 7240
 7270 PRINT d$+cl$+phyle$
 7280 GOSUB 3900
 7290 RETURN
 7300 POP: RETURN
 7398 REM     play
 7400 t = 1
 7410 a$ = note$(t)
 7420 IF a$ <> "z" THEN  GOSUB 2000: t = t+1: GOTO 7410
 7430 sound%(t, 0) = -1
 7440 GOSUB 5000
 7450 RETURN
 7498 REM  delete a note
 7500 HOME: VTAB 10: HTAB 2: PRINT "What is the number of the"
 7510 HTAB 5: PRINT "note you want to delete?"
 7520 HTAB 5: INPUT "enter it and press return "; td
 7530 i = td+1
 7540 note$(i-1) = note$(i)
 7550 IF note$(i) = "z" THEN 7580
 7560 i = i+1
 7570 GOTO 7540
 7580 HOME: VTAB 10: HTAB 3
 7590 PRINT "The note has been deleted": NORMAL
 7600 FOR i = 1 TO 5000: NEXT i
 7610 RETURN
 7698 REM  create the editing screen
 7700 HOME: VTAB 2: HTAB 2
 7710 PRINT "OPERATION:"; SPC(4); "NOTE NUMBER:"
 7720 VTAB 9: HTAB 2
```

(continued)

```
 8310 HTAB 2: PRINT "to insert the new note?"
 8320 HTAB 2: PRINT "Enter it then press return."
 8330 INPUT t8
 8340 t = t8
 8350 t = t+1
 8360 IF note$(t) = "z" THEN 8380
 8370 GOTO 8350
 8380 FOR i = t TO t8 STEP -1
 8390 note$(i+1) = note$(i)
 8400 NEXT i
 8410 up$ = "Insert"
 8420 GOSUB 7700
 8430 RETURN
 8698 REM   enter menu
 8700 HOME: VTAB 10: HTAB 5: PRINT "Do you want to enter:"
 8710 HTAB 3: PRINT "I: A new piece"
 8720 HTAB 3: PRINT "II: Add to the current one"
 8730 HTAB 3: PRINT "III: exit this menu"
 8740 VTAB 18: HTAB 5: PRINT "Press that SmartKey"
 8750 GET ch$: ch% = ASC(ch$)-128
 8760 IF ch% > 3 OR ch% < 1 THEN  GOSUB 300: GOTO 8700
 8770 ON ch% GOSUB 8900, 9200, 7300
 8780 GOTO 8700
 8898 REM  entering a new piece
 8900 HOME: VTAB 10: HTAB 2: PRINT "In entering a new
selection"
 8910 HTAB 2: PRINT "there are no defaults for the"
 8920 HTAB 2: PRINT "first note"
 8930 FOR i = 1 TO 6000: NEXT i
 8940 t8 = 0: up$ = "enter"
 8950 GOSUB 7700
 8960 IF note$(t8+1) = "z" THEN  RETURN
 8970 t8 = t8+1
 8980 GOTO 8950
 9198 REM  append to a selection
 9200 t = 0
 9210 t = t+1: IF note$(t) = "z" THEN 9230
 9220 GOTO 9210
 9230 t8 = t-1
 9240 up$ = "append": GOSUB 7700
 9250 IF note$(t8+1) = "z" THEN  RETURN
 9260 t8 = t8+1
 9270 GOTO 9240
 9398 REM  defaults for notes
 9400 IF t8 = 0 THEN  RETURN
 9410 t = 0: a$ = note$(t8): GOSUB 2000
 9420 IF oct$ = "" THEN  oct$ = STR$(oct%)
 9430 IF loud$ = "" THEN  loud$ = STR$(loud%)
 9440 IF dur$ = "" THEN  dur$ = STR$(dur%)
 9450 IF voice$ = "" THEN  voice$ = STR$(voice%)
 9460 IF nn$ <> "" THEN  RETURN
```

(continued)

```
 9470 IF MID$(a$, 5, 1) = "#" THEN  np$ = MID$(a$, 4, 2)
: GOTO 9490
 9480 np$ = MID$(a$, 4, 1)
 9490 nn$ = np$
 9500 RETURN
 9998 REM      data
10000 DATA     27450,58,64, 107,211,255,201
10010 DATA     32.703,36.708,41.204,43.654,49.,55.,61.735
,34.648,38.892,46.249,51.915,58.32
```

While you're debugging the program, it's possible that the program will turn the sound on but you won't be able to turn it off. The easiest solution is to turn the sound down until the next time you run the program. Otherwise, the only way to shut off the sound is to reset ADAM. (The sound can also be shut off using another program - which consists of the DATA statement 10000, the machine language program and the turnoff subroutine.) Once the program is debugged, each note is automatically turned off after it has been played.

Just as for the ShapeMaker, once the *Smart Musician* is debugged it's a good idea to store it on a (relatively) freshly INITed data pack. This will speed up the loading of the program.

Using the Smart Musician

The *Smart Musician* program is a music editor and translator. It lets you write music in special codes (see below). One part translates these codes into instructions for the sound generator and another tells the sound generator to play the notes. After you enter the codes, ADAM will translate all of them. After the translation is complete you can save the piece or play it or both. ADAM must completely translate the notes before playing the piece. Otherwise ADAM will stop the music after each note and translate the next one.

After starting to write the music using the editor, you can tell ADAM to file the part you've entered and use the word processor to enter the rest. This is only useful when you have to delete many notes. (In order to listen to the piece again, you'll have to reload SmartBASIC and the *Smart Musician* program. This takes quite a while.)

Each note consists of five pieces of information.

1. the octave of the note

2. the note itself
3. the duration
4. the volume (loudness)
5. the voice

These may need a little explanation. (To learn how to read music you might look at the Self-Teaching Guide, *What Makes Music Work* by Seyer, Novick and Harmon.) There are twelve notes in each octave C,C#,D, D#,E,F,F#,G,G#,A,A# and B. The note C# is called C-sharp, D# is D-sharp, etc. These notes repeat in each octave. So you need to know the octave, the note's name and the duration to specify it completely. (Sometimes you'll see notes like A-flat A♭ but this is also a G#. The *Smart Musician* recognizes only sharps.) Most pieces of music sound best if you tell ADAM to play the note one octave higher than it's usually given in sheet music.

The duration of a note is measured in fractions of a *whole* note. The smallest fraction we allow is 1/64. You tell ADAM the length of a note by entering the number of 64'ths of a whole note. For example a quarter note is coded as 16, an eighth note is 8, a whole note is 64, etc.

The volume or loudness is a number from 0 to 15. Zero is silent and acts as a pause and 15 is the loudest.

The voice is one of the numbers 1, 2 or 3.

ADAM first stores a note as a string

O____ N____ d____ L____ V____

Each underline is replaced by one of the entries mentioned above. When you use the editor, you are prompted for each entry with a question mark next to the name of the entry. You can give the numbers or notes, then ADAM will form the string. If you hit RETURN after a prompt, ADAM will use the same entry as for the previous note. Of course, this can't work for the first note. In other words, the default value of any part of a note code is the code for the corresponding part of the previous note.

If you follow the same form you can use other programs to write the notes. After storing them in a file, the *Smart Musician* can play them for you. For example, you might write a program to compose harmonies or counterpoints. When you use another program to write the notes, you must

store the note in exactly the forms shown above. The program uses slightly different defaults if you use a different program to write the notes. Each octave and note must be specified; but the duration and loudness default to the entry for the previous note. The voice defaults to the first voice. In any other program that writes music to feed to the SmartMusician it's safest to specify the note completely.

Using the Program

When the program is running it gives almost all the directions you'll need. After you see the title display, you press any key to go to the main menu. It looks like:

MAIN MENU

I. Write some music

II. Play from a data pack - an untranslated piece

III. Play from a data pack - a translated piece

IV. Replay whatever is in memory

V. End the program

You make your choice by pressing a SmartKey. If you ask to play an untranslated piece, ADAM asks whether you want to save the translation. Since the translation takes a little while, it's best to save the translated version and use it to play the piece again. However, the translation isn't readable anymore - it's in code. If you save the untranslated version you can look at the music or change it. The translated version is the source code (*source code* is the original, readable version of a program.) We might store the original, untranslated source code as:

tune

The translated version is stored in object code. (*Object Code* is the translated version of a program). We use a name like:

tune.t

so the .t reminds us its been translated.

The first thing you probably want to do is to write some music, so you press SmartKey I. You get the main editing menu:

MAIN EDITING MENU

I. Enter

II. Insert

III. Delete

IV. Play

V. Save the untranslated version

VI. Back to the main menu

You can insert a note after any note already in memory as well as delete any note. One way to check if you want to change a note is to play the piece in memory slowly. When you play it, the number of each note is displayed on the left hand side of the screen. Jot down the number of the note you want to change. Be careful though, the fastest way to bomb the program is to try to delete a note that doesn't exist or to delete the end-of-music flag. The flag is the single character "z".

To start writing some music, press SmartKey I again. This sends you to the ENTER menu.

ENTER MENU

I. Enter a new piece

II. Append to the end of the piece in memory

III. Exit this menu

Append enters notes at the end of the piece in memory. (The end-of-data flag is erased.) You cannot append at the first note - that's writing a new piece. If you enter a long selection, save whatever you've entered every ten minutes or so. This way if ADAM or the program crashes, you won't lose everything. To load a selection into memory, go back to the main menu and press SmartKey II. Then go to the editing menu to add to it. Whenever you write a new piece, append to an existing piece or insert a note, ADAM displays the *editing screen*. The screen shows you:

1. each of the entries you must make to form an instruction.

2. the number of the instruction you are writing

3. the five previous notes (if there are five)
4. some instructions

A flashing question mark appears next to

octave =

Enter the octave number and press RETURN. The question mark will move down to

note =

and you need only respond to the prompt (and hit return). Continue following the prompts until you have told ADAM which voice to use. To play two or three notes simultaneously, use voice 1 for the last note entered.

When you press RETURN after entering the voice, the whole note is displayed near the bottom of the screen. If it is correct, hit the space bar; if not, hit any other key. The *Smart Musician* gives you another chance to enter the note correctly. After you hit the space bar, you automatically start to write the next note. To stop entering notes, answer the **octave** = ? prompt with the letter z. ADAM will return to the enter menu and then pressing SmartKey III, returns ADAM to the main editing menu.

The main editing menu lets you choose to play the piece you've just written, save it or go back to the main menu. Once you're at the main menu you can end the program.

Here's an example of *Auld Lang Syne* (the traditional New Year's Eve song) written in this language. The entries (notes) are arranged so that you enter each row in the first column, then go to the second column. If no entry appears, press RETURN because it's the same as the previous one.

Auld Lang Syne

O	N	d	L	V
3	C	8	15	1
	F	24		
	F	16		
	F			
	A			
	G	24		
	F	16		
	G			
	A			
	F	24		
		16		
	A			
4	C			
	D	24		
	F	8		

O	N	d	L	V
	C	24		
3	A	16		
	A			
	F			
	G	24		
	F	16		
	G			
	A			
	G			
	F	24		
	D			
	D			
	C			
	F	24		

Actually this isn't the whole song - it's just one stanza. Another piece that sounds nice on ADAM is the following part of a minuet by Bach (from the Anna Magdelena Bach Notebook):

O	N	d	L	V
4	D	16	15	1
3	G			
	A			
	B			
4	C			
	D			

O	N	d	L	V
3	G			
	G			
4	E			
	C			
	D			
	E			

O	N	d	L	V
	F#			
3	G			
	G			
4	G			
	C			
	D			

O	N	d	L	V
	C			
3	B			
	A			
	B			
4	C			
3	B			
	A			
	G			
	F			
	G			
	A			
	B			
	G	8		
	A	48		
4	D	16		
3	G			

O	N	d	L	V
	A			
	B			
4	C			
	D			
3	G			
	G			
4	E			
	C			
	D			
	E			
	F#			
	G			
3	G			
	G			
4	C			
	D			

O	N	d	L	V
	C			
3	B			
	A			
	B			
4	C			
3	B			
	A			
	G			
	A			
	B			
	A			
	G			
	F			
	G	32		

To transfer an untranslated selection from one data pack to another, do the following:

1. In the main menu, play the selection
2. When the play is over, don't save it.
3. From the main menu, select the main editing menu
4. Change data packs
5. Save the untranslated version

Transferring a translated version is easier. Play it from the main menu, change data packs and save the translated version.

SOLUTIONS TO SELF-CHECKS

Chapter 3

Self-Check 1

B.
1. Commas are not allowed in numbers.
2. PRINT is mispelled.
3. **X** is not allowed as a symbol for multiplication.
4. Same as 3.
5. Division by 0 is not permitted.

Self-Check 2

1. 65,536
2. 1,024
3. Will accept 3__80
4. Will accept 3__−80
5. 81,100

Self-Check 3

A. 1. −3.0568E4
2. 5.4E−4
3. 1.0054E−1
4. 2.3E−10

B. 1. 105
2. −.00000036207
3. .00021
4. 860,000,000,000,000

Self-Check 5

A. 5. This problem asks you to compute the square root of a negative number.

Self-Check 6

A. The table entries reading across are 10.60, 16.16, 22.09

B. The fifteen inch pizza is cheapest per square inch.

Chapter 4

Self-Check 1

A. It prints on different lines

```
HE          L

HELP
```

Self-Check 2

A. ADAM will not accept
2. "ADAM" is a string and can't be assigned to the variable **a**.
3. The final quotation mark is missing.
4. The first quotation mark is missing.

Self-Check 3

A. 1. A number (or numeric) variable can't be string.
3. Arithmetic must be performed on the right-hand side of an assignment statement.

The others are correct

B. Prints
22, 10, 12
22, 12, 12

Chapter 5

Self-Check 1

A. 1. Illegal - no line number.
2. Illegal - # not allowed.
3. Illegal - no space is allowed in GOTO.
4. Same as 2.
5. Illegal - no comma is allowed in line numbers (the number is also too large).
6. This is not the proper way to put two statements on one line.
7. is legal
8. Variables are not allowed in GOTO's.
9. Same as 8
10. Strings are not allowed - these must be line numbers.
11. is legal
12. is legal but dangerous and silly

B. The program is trapped in an infinite loop.

C. Add up the numbers from one to the number entered.

E. The display only stands still when the number of (characters counting spaces) is a multiple of 31.

Self-Check 3

A. The program will print the numbers twice. Once correctly and once not. The program can be fixed in one of two ways. Either replace line 20 by:

IF *a*$ < *b*$ THEN PRINT *a*$,*b*$: GOTO 40

or replace line 30 by:

IF *a*$ > *b*$ THEN PRINT *b*$,*a*$

Self-Check 4

A.

1.

d) is wrong. The THEN clause is illegal.

e) is wrong. It is impossible to test whether a string is equal to a number.

g) is wrong. The IF clause must precede the THEN clause.

Self-Check 5

A.

1.

a) No. The variable *a* must be repeated in the IF clause.

2.

a) This program only ends when a number less than 0 is entered.

b) Ends when the word no is entered twice.

Self-Check 6

A. 1. No, the variable *c* is never defined

2. No. It would if you added line 25 LET *c* = *ws*

3. Still no. You need a test for a terminator to work.

Chapter 6

Self-Check 1

A. 1. No NEXT *i*

2. Prints the numbers from 99 to 0

Self-Check 2

A. 1. About 27 seconds

Self-Check 3

A. 1. a) twice

b) once

c) once (any FOR-NEXT loop runs once!)

d) as in c)

e) Forever

2. 3 = starting value
   15 = test value
   1 (=4−3) is the step value.

Self-Check 4

A. 1. The speed wasn't changed back to 255
2. The variable *n* is never assigned a value and the loops are nested incorrectly.

Chapter 7

Self-Check 1

A. 1. in the right bottom corner.
2. about the middle of the first row.
3. two thirds of the way down the last row.
4. in the left bottom corner.

Self-Check 2

A. 1. Place a vertical line in column 20 from row 3 to row 33
2. Place a horizontal line in the bottom row

Self-Check 3

A. 1. Two units in value.
2. About four units in value.
3. You have to divide the largest value by 39 to scale the data.
4. They could be too large to plot.

Self-Check 4

A. 1. You would get an error message. You can only use the SCRN command for values between 0 and 39.
2. Print out the current values of col and row.
3. ADAM responds with "TYPO. Retry".
4. ADAM is still in GRAPHICS mode.

Chapter 8

Self-Check 1

A. 1. The first separates the words by 10 spaces, the second prints GOODBYE starting at the 10th column.

Self-Check 2

A. 1. At the end of the middle line on the screen.
2. Yes. The first command tells ADAM to print the dollar sign in the first column of the last row. The second command tells it to print the dollar sign in the last column of the current row.

Self-Check 3

A. 1. REM prints the lower case alphabet

Chapter 9

Self-Check 1

A. 1. Between 0 and 99.9999999

Self-Check 2

A. 1. −48, 45, 47
2. −22
3. 1
4. Integers between 0 and 999

Self-Check 3

A. 1. 21 − INT(21/5)*5
2. −21 + INT(21/5)*5
3. 149854 − INT(149854/5)*5

Self-Check 4

A. 1. DEF FN odd(x) = x − INT(x/2)*2
2. DEF FN re(x) = x − INT(x/10)*10
3. DEF FN deg(x) = x*180/4*ATN(1)

Chapter 10

Self-Check 1

A. 1. The method the person with i.d. number 4 uses to get around town.
2. A string list with 126 entries. A list with 13 numeric entries.
3. A string list has been dimensioned but a numeric list is being used.

Self-Check 2

A. 1. READ c means read a number, hello isn't one.
2. The counter would be off by one.

Self-Check 3

A. 1. The program compares entry *x(j)* with an entry past it on the list. The (n−1)st entry is the last one with something past it.
2. What good is it to compare an entry with itself?
3. Change all the < to >.

Chapter 11

Self-Check 1

A. 1. Fall weather in Paris.
2. 18 (or 48 if you use zero as an index value).
3. The row index (35) exceeds that given in the DIMension statement.

Self-Check 2

A. 1. *able* - real variable
able$ - string variable
able% - integer variable
2. ADAM would paint a sky blue block in the center of the graphics screen.

Self-Check 3

A. 1. a. (150,84)
b. (24,86)
c. (150,−18)
2. a. (20,−8)
b. (−20,−30)
c. (−40,70)

Self-Check 4

A. 1. Usually scaling is used to make numbers of sizes of things manageable. For example, scaling is used to make graphs fit on the display screen.
2. a. 1/4 inch
b. 1000 inches (83 1/3 feet)
c. about .0947 inches
3. So that all entries are larger than the starting value. If $max = 50$ and all the entries were less than fifty, the scaling module would scale the data as though fifty was the size of the biggest entry.
4. Multiply each data entry by $-sc$.
5. Subtract the reference value from the maximum entry.

Chapter 12

Self-Check 1

A. 1. There is no END statement before the subroutine.
2. global variable *q*
local variable *q*

Self-Check 2

A. 1. A top-down program is designed to have different levels. You must make sure that information only passes in the correct direction. Requests for action pass down. Results of operations pass up.
2. In the program on page 284
global variables - *st,tst,tep*
local variable - *i*

The whole problem with the program on page 288 is that *s* is used in both ways. It is global because it passes between levels and it is local because it is changed at a lower level in the pyramid.

Self-Check 3

A. 1. They allow functions with more than one variable.
2. This is done so that the value is always recalculated.
3. You get an error message.
4. Same as 3.

Chapter 13

Self-Check 1

A. 1. The answer is "ADAMand EVE"
2. Change the counter to run from 65 to 90.
3. 12

Self-Check 2

A. 1. 14
2. Change line 1020 to
IF m7 > 80 . . .
Change line 1030 to read
PRINT SPC(80−m7/2); c7$
Omit lines 1040 and 1050
3. HELLO
GOODBYE
7 2

Self-Check 3

A. 1. "I am"
2. "am a"
3. "I'm a friendly computer"
4. "m a friendly computer"
5. "I'm a friendly computer"

Self-Check 4

A. 1. a$ = MID$(a$,1,3) + MID$(a$,5) is the obvious answer but it

won't work if *a$* has fewer than 5 characters. The following two lines of code always work:

```
IF LEN(a$) = 4 THEN a$ = MID$(a$,1,4)
IF LEN(a$) > 4 THEN a$ = MID$(a$,1,3)+MID$(a$,5)
```

When LEN(a) < 4, we want *a$* to remain unchanged. In that case, these two lines don't change *a$*.

2. "HE"
3. "BYE"

Self-Check 5

A. 1. The GET command accepts only one character and *doesn't* require hitting RETURN.
2. If *key* = 0, then we don't want to switch the words. We have ADAM move to the next word in the list.

Self-Check 6

A. 1. "123.2"
2. Yes, 46

Chapter 14

Self-Check 1

A. 1. 256 columns by 159 rows
3. Write the HPLOT in a subroutine. To alternate the colors, feed the subroutine first HCOLOR = 2, then HCOLOR = 14, then HCOLOR = 2, etc.

Self-Check 2

A. 1. Use

```
HPLOT x1,y1 TO x2,y2 TO x3,y3 TO x4,y4
```

2. *HPLOT row, column* plots a single dot
HPLOT TO *row,column* plots a line from the last point drawn to the indicated location on the screen.
HPLOT *row1,column1* TO *row2,column2* connects the two indicated points by a line.
In the last use of the command, you can keep adding TO's and extra locations (see the answer to the previous problem).

Self-Check 3

A. 1. In HGR2, you lose the TEXT window. The screen has more rows (192).
2. You won't be able to see any TEXT until ADAM leaves HGR2 mode and returns to TEXT.

Self-Check 4

A. 1. It responds more quickly.
2. It tells ADAM whether the left firing button on the front game paddle has been pressed.

Chapter 15

Self-Check 1

A. 1. It rotates the shape 45° .
2. SCALE = 0 blows up the shape. It looks very strange.

Self-Check 2

A. 1. 10% (90,24)
50% (50,80)
75% (75,115)

Chapter 16

Self-Check 1

A. 1. A backup copy saves you the trouble of rewriting a program or regathering data in case the original copy is lost, destroyed or becomes unusable.
2. The code letter **a** means an earlier version of a program than the one with letter **a**. It means that you did not delete the older version before saving the new one.
3. Suppose you have two files named *phyle*, one with code letter **a** and the other with letter **A**. If you RENAME *phyle* and then RECOVER, you get to keep the latest version of *phyle* under a different name. If you just delete it, it is lost forever.

Self-Check 2

A. 1. You'll get an error message.

2. ADAM thinks that the file uses all the remaining spaces on the data pack. The data pack cannot store any new files. If you do this, reinsert the data pack in the drive, look at the CATALOG to find the name of the unclosed file (it's the one that is extremely long). In direct mode enter

CLOSE *filename*

3. ADAM will bomb because it can't find the file to CLOSE. The file will also remain open on the data pack and you should look at the directions given in the answer to question 2.
4. If you reset ADAM without the SmartBASIC data pack in the drive, ADAM will go back to being an electronic typewriter. Press the **WP** key to go into the word processor. When you press the **GET** key and follow the directions, you can have ADAM display (almost) any file.

Self-Check 3

A. 1. The error condition with code number 42 is the OUT OF DATA condition. It means that ADAM has tried to READ more entries than there are in a DATA list.
2. CLRERR cancels the ONERR GOTO command.
3. The main advantage of the ONERR GOTO command is to give your program a chance to patch errors before ADAM automatically bombs a program. The disadvantage is that your program can give meaningless results when error conditions, that should bomb a program, are covered up by the ONERR GOTO.

Self-Check 4

A. 1. use the commands:

PRINT *d*$; "DELETE "; *phyle*$
PRINT *d*$; "RENAME "; *phyle*$

2. Integer variables take up less space in memory than real variables. In addition integer arithmetic is usually faster than arithmetic using real variables. *a*% is a long list, so it pays to

save space. *ky%* is one number, used once. It doesn't make much difference whether it is integer or real.

3. Each file entry is an ASCII code plus a positive number. It can't be negative.

Self-Check 5

A. 1. WRITE doesn't OPEN a file. It set the pointer to the beginning of a file and will overwrite anything in the file. APPEND OPENs a file and sets the pointer after the last entry in the file. It doesn't overwrite anything in the file.

2. *count* starts at 0 and increments before the test.
kount starts at 1 and increments after the test.

APPENDIX A
The ASCII Codes

Appendix A is divided into four parts. Part 1 lists the display symbols and ASCII codes for the usual control keys. Part 2 lists the functions of the different control key combinations. This list also includes a few non-standard control key combinations that have higher order ASCII codes. Part 3 lists the (usual) ASCII codes. These are the ones from 32 through 127. Part 4 lists the *higher order ASCII codes* peculiar to ADAM.

Part 1

Display symbols and ASCII codes for control keys

ASCII CODE	KEY (COMBINATION)	DISPLAY SYMBOL	ASCII CODE	KEY (COMBINATION)	DISPLAY SYMBOL
0	CONTROL/@		16	CONTROL/P	
1	CONTROL/A		17	CONTROL/Q	
2	CONTROL/B		18	CONTROL/R	
3	CONTROL/C		19	CONTROL/S	
4	CONTROL/D		20	CONTROL/T	
5	CONTROL/E		21	CONTROL/U	
6	CONTROL/F		22	CONTROL/V	
7	CONTROL/G		23	CONTROL/W	
8	CONTROL/H		24	CONTROL/X	
9	CONTROL/I		25	CONTROL/Y	
10	CONTROL/J		26	CONTROL/Z	
11	CONTROL/K		27	CONTROL/[	
12	CONTROL/L		28	CONTROL/\	
13	CONTROL/M		29	CONTROL/]	
14	CONTROL/N		30	CONTROL/^	
15	CONTROL/O		31	CONTROL/_	

Part 2

The Uses of Control Keys

CONTROL/C	Halts program processing
CONTROL/D	Used with a PRINT statement to direct output to, or receive output from a peripheral device.
CONTROL/G (*)	Rings the bell
CONTROL/H	Acts as a backspace
CONTROL/I	Acts as a tab
CONTROL/J	Forces a line feed
CONTROL/K (*)	Vertical tab
CONTROL/L	Clears screen (same as HOME command)
CONTROL/M (*)	Carriage return and line feed
CONTROL/N	Adds a space, CAUTION when used in editing mode acts only on the physical line. If used in the form of PRINT CHR$(14) when the printer is activated will send the printhead running backwards.
CONTROL/O	Deletes a character, CAUTION when used in editing mode acts only on physical line. If used in the form of PRINT CHR$(15) when the printer is activated will send the printhead running forward again.
CONTROL/P	Prints the screen
CONTROL/S	Suspends processing (including the printer) until any key is hit.
CONTROL/X	Causes ADAM to forget the line being typed. Prints a backslash \ to indicate this.
CONTROL -->	This is ASCII code 165. When used to edit a line this causes the character the cursor is under not to be read.
CONTROL <--	This is ASCII code 167. When used to edit a line this causes the character the cursor is under not to be forgotten.

The control key combinations marked with a * must be used in the form of PRINT CHR$( ) to activate the specified function.

Part 3

The standard ASCII codes

The characters from 32 to 128 are standard on almost all microcomputers.

ASCII CODE	DISPLAY SYMBOL (AND KEYSTROKE)	ASCII CODE	DISPLAY SYMBOL (AND KEYSTROKE)
32	SPACE BAR	70	F
33	!	71	G
34	"	72	H
35	#	73	I
36	$	74	J
37	%	75	K
38	&	76	L
39	'	77	M
40	(	78	N
41	)	79	O
42	*	80	P
43	+	81	Q
44	,	82	R
45	-	83	S
46	.	84	T
47	/	85	U
48	0	86	V
49	1	87	W
50	2	88	X
51	3	89	Y
52	4	90	Z
53	5	91	[
54	6	92	\
55	7	93	]
56	8	94	^
57	9	95	_
58	:	96	`
59	;	97	a
60	<	98	b
61	=	99	c
62	>	100	d
63	?	101	e
64	@	102	f
65	A	103	g
66	B	104	h
67	C	105	i
68	D	106	j
69	E	107	k

ASCII CODE	DISPLAY SYMBOL (AND KEYSTROKE)	ASCII CODE	DISPLAY SYMBOL (AND KEYSTROKE)
108	l	118	v
109	m	119	w
110	n	120	x
111	o	121	y
112	p	122	z
113	q	123	{
114	r	124	|
115	s	125	}
116	t	126	~
117	u	127	DELETE

Part 4

Codes Peculiar to ADAM

The ASCII codes higher than 128 are different on each computer. For ADAM these are used as the codes for the extra keys as well as to display reverse video.

ASCII CODE	KEY (COMBINATION)	DISPLAY SYMBOL	ASCII CODE	KEY (COMBINATION)	DISPLAY SYMBOL
128	Home Key		152	Shift & Wild Card	
129	Smart Key I		153	Shift & Undo	
130	Smart Key II		154	Shift & Move/Copy	
131	Smart Key III		155	Shift & Store/Get	
132	Smart Key IV		156	Shift & Insert	
133	Smart Key V		157	Shift & Print	
134	Smart Key VI		158	Shift & Clear	
135			159	Shift & Delete	
136			160	Up arrow	
137	Shift & Smart Key I		161	Right arrow	
138	Shift & Smart Key II		162	Down arrow	
139	Shift & Smart Key III		163	Left arrow	
140	Shift & Smart Key IV		164	Control and up arrow	$
141	Shift & Smart Key V		165	Control and right arrow	%
142	Shift & Smart Key VI		166	Control and down arrow	&
143			167	Control and left arrow	'
144	Wild Card		168	Up arrow and right arrow	(
145	Undo		169	Down arrow and right arrow	)
146	Move/Copy		170	Down arrow and left arrow	
147	Store/Get		171	Up arrow and left arrow	+
148	Insert				
149	Print				
150	Clear				
151	Delete				

ASCII CODE	KEY (COMBINATION)	DISPLAY SYMBOL
172	Home and up arrow	
173	Home and right arrow	
174	Home and down arrow	
175	Home and left arrow	
176		0
177		1
178		2
179		3
180		4
181		5
182		6
183		7
184		8
185	Shift and TAB	9
186		:
187		;
188		<
189		=
190		>
191		?
192		@
193		A
194		B
195		C
196		D
197		E
198		F
199		G
200		H
201		I
202		J
203		K
204		L
205		M
206		N
207		O
208		P
209		Q
210		R
211		S
212		T
213		U
214		V

ASCII CODE	KEY (COMBINATION)	DISPLAY SYMBOL
215		W
216		X
217		Y
218		Z
219		
220		
221		
222		
223		
224		`
225		a
226		b
227		c
228		d
229		e
230		f
231		g
232		h
233		i
234		j
235		k
236		l
237		m
238		n
239		o
240		p
241		q
242		r
243		s
244		t
245		u
246		v
247		w
248		x
249		y
250		z
251		{
252		|
253		}
254		~
255		
256		

APPENDIX B
Error Codes and Messages

0 NEXT WITHOUT FOR

You've left out the FOR command that begins a loop or have incorrectly nested your loops.

2 RANGE ERROR

You've used some number in an operating system command that is bigger than ADAM can handle. For example, a request for information from drive 100.

5 END OF DATA

Your program tries to READ more information than is contained in a data file.

7 FILE NOT FOUND

Your program asks ADAM to use a file that it can't find. The file name you entered may not be spelled the same as in the CATALOG. Mixing up lower and upper case letters is a common problem. Enter the command CATALOG to check the required spelling. This error message can also appear when a data pack has become unusable or the tape heads are dirty or misaligned - try cleaning them, alignment requires professional servicing.

8 I/O ERROR

ADAM can't read or write to a data pack. There may not be one in the drive, the one there may be in backwards or there may be a hardware failure.

9 NO MORE ROOM

There are three possible causes for this error condition. You may already have stored 35 files on the data pack. You may have used all 253 blocks on it. Most likely you left a data file OPEN.

* Some of the error codes are used for more than one error message. The error codes are the same as in Applesoft, but ADAM's messages are more descriptive. (See Appendix D.)

10 FILE LOCKED
Your program tried to WRITE to a locked file. The file must be UNLOCKED before it can receive any output.

11 SYNTAX ERROR
In an operating system command, you've used something illegal, for example, punctuation or an illegal file name.

12 CONTROL BUFFER OVERFLOW
More characters have been sent into the control buffer than it can handle. (The control buffer holds the operating system commands.)

12 NO BUFFERS AVAILABLE
You've tried to have more than two data files open at the same time.

13 FILE TYPE MISMATCH
You tried to run a binary file as though it were a SmartBASIC program file or something similar, for example, you tried to run a data file as though it contained a program.

16 SYNTAX
The statement that you've entered is not permissible in SmartBASIC. Typical errors are missing parentheses and missing or extra characters.

22 RETURN WITHOUT GOSUB
Your problem gets to a RETURN statement without being sent into a subroutine by a GOSUB command. ADAM doesn't know where to return to.

42 OUT OF DATA
ADAM is trying to READ from a DATA list and all the entries have already been read. Either there is too little DATA or your program is trying to READ too much.

53 ILLEGAL QUANTITY
You've tried to use a quantity or string that is outside the valid range or not permissible for the intended use. Examples are: asking ADAM to plot outside the graphics screen, trying to use MID$ on a number instead of a string.

69 OVERFLOW
You've made a calculation that results in a number too big for ADAM to handle.

77 OUT OF MEMORY
This message appears when any of the following error conditions occurs: the program is too large to fit in memory; your program uses too much space to store variables, uses too many nested FOR-NEXT loops, subroutines or parentheses. LOMEM and HIMEM may be set too close together, resulting in too little space to hold the program and variables.

77 STACK OVERFLOW
The stack doesn't have room for all the FOR-NEXT loops, subroutine return locations or parentheses that your program uses at the same time.

90 UNDEFINED STATEMENT
Your program branches to a line that isn't in your program.

107 BAD SUBSCRIPT
The dimensions of the array do not permit the subscript used.

120 REDIMENSIONED ARRAY
Your program tries to dimension the same array (at least) twice. This is not legal without a CLEAR command between the two attempts. This can also occur if an array is used before it is dimensioned (The default dimensioning is invoked by ADAM. The DIMension statement then tries to redimension the array).

133 DIVIDE BY ZERO
Computers are no better than people at dividing by zero.

163 TYPE MISMATCH
You've used a numeric expression where a string is expected or vice versa.

176 STRING TOO LONG
You've constructed a string that has more than 255 characters (probably by using concatenation).

224 UNDEFINED FUNCTION
Your program uses a function before it defines it.

254 REENTER
ADAM is not happy with your response to an INPUT request. You may have entered a string when a number is required or something similar.

255 BREAK
This is not an error. The code appears as the value of ERRNUM(1) when an error trap has been set and you try to stop a program using the command STOP.

(no code) CAN'T CONTINUE
The command CONT tells ADAM to continue processing a program that has stopped for some reason. This message indicates that the program cannot continue due to a change in the program or another error condition. It also appears when you stop listing a program or printing the CATALOG and try to continue.

(no code) FATAL SYSTEM ERROR
Your program has an error that ADAM can't handle at all. It can't even give an error code. You should press the reset button and reload SmartBASIC because SmartBASIC or the operating system, in RAM, may have been overwritten by garbage.

(no code) ILLEGAL MODE
You've tried to use a command in immediate mode that can only be used in programming mode or vice versa.

APPENDIX C
Mathematical Formulas

The following list gives some of the most commonly used formulas, functions and constants. SmartBASIC doesn't have all possible mathematical functions built in; but it's easy to use this list to make your own *user defined functions*. [If you use these formulas to make your own user defined functions, remember only the first two letters count.]

pi = 4*ATN(1)	(π about 3.1415927)
e = esp(1)	(e about 2.7182818)

In the formulas that follow we've used *pi* as defined above as our approximation to π. [Alternatively you can use 3.1415927.]

degrees to radians	rad = deg * 180/pi
radians to degrees	deg = rad * pi/180

sec(x) = 1/COS(x)

csc(x) = 1/SIN(x)

cot(x) = 1/TAN(x)

arcos(x) = = ATN(X/SQR(−X*X+1) + pi/2

arcsin(x) = −ATN(X−SQR(−X*X+1))

arccot(x) = − ATN(X) + pi/2

arcsec(x) = ATN(SQR(X*X−1))+(SGN(X)−1)*pi/2

arccsc(x) = ATN(1/SQR(X*X−1))+(SGN(X)−1)*pi/2

cosh(x) = (EXP(x) − EXP(−x))/2

sinh(x) = (EXP(x) − EXP(−x))/2

tanh(x) = 1 + 2*(EXP(−x)/(EXP(x) + EXP(−x)))

arcsinh(x) = LOG(X + SQR(1+X*X))

arccosh(x) = LOG(X + SQR(X*X−1))

arctanh(x) = LOG((1+X)/(1−X))/2

$\log_{10}(x) = LOG(X)/LOG(10)$ common logarithum

$\log_a(x) = LOG(X)/LOG(a)$

$\text{mod}_a(x) = INT(X - INT(X/a)*a)$ for $a > 0$

APPENDIX D
The Difference Between SmartBASIC and Applesoft BASIC

SmartBASIC was designed to recognize and execute Applesoft BASIC commands, that is, the version of BASIC run on the Apple II series of microcomputers. This was a wise decision by Coleco because more personal and educational programs are written in Applesoft than in any other version of BASIC. Unfortunately, not every Applesoft program entered into ADAM will run without change. There are several reasons for this. The most important is that Apple computers — running Applesoft — use versions of the 6502 microprocessor as their CPU, while ADAM uses a Z80A (see Chapter 2). These two chips are quite different. The screen displays and sound also work differently on ADAM and on the various Apples. In this appendix, we'll explain the differences as well as how to know if it's easy to translate a program from Applesoft to SmartBASIC. Finally we'll show you the key signs that will tell you when translation requires a major effort.

Screen Display - Text

The first difference between ADAM and the APPLE is that the text screens are different. The Apples have 40 or 80 column displays while ADAM has 31. This means that almost every carefully formatted Applesoft screen display looks weird on ADAM. You can either accept the resulting display or rewrite the commands in the program that control the screen displays.

The number of rows on each machine is the same.

Screen Display - Graphics

Both ADAM and the Apple have the same graphics commands and the same 40 by 40 screen in low resolution graphics. However the high resolution screens have different sizes. The Apple HGR screen is 280 columns by 160 rows; the HGR2 screen is 280 by 192. The corresponding numbers for ADAM are 256 by 160 and 256 by 192. An Applesoft program that uses columns numbered above 255 will bomb when run on ADAM. You can rescale the display to make the translation work. It's easiest to use the scale factor 255/279 for the column positions.

One graphics command does work slightly differently in the two versions of BASIC. In SmartBASIC the command XDRAW must be preceeded by the DRAW command for the same shape (from a shape table). In Applesoft this is not necessary. Since XDRAW is usually used to erase a shape in animation, this difference has little effect.

Finally, ADAM has more colors than the Apple in high resolution graphics. The HCOLOR command must be changed to reflect this. Here's a list of Apple high resolution color codes.

code number	color
0	black
1	green
2	violet
3	white
4	black
5	orange
6	blue
7	white

Error Traps

The commands ONERR____GOTO, CLRERR and RESUME are the same in both languages. Applesoft doesn't have the ERRNUM(1) command. The error number is stored in memory location 222. So each time you see the command

```
a = PEEK(222)
```

substitute

```
a = ERRNUM(1)
```

Otherwise error trapping is the same.

Speed of the Machine

As mentioned above, ADAM runs faster than the Apple. For example, a timing loop that takes 10 seconds on the Apple, may take only five or six on ADAM. If you see a timing loop in an Applesoft program, try multiplying the test value by 1.5 in the translation.

Many Applesoft programs use PEEKs and machine language routines to speed up program execution. Since ADAM runs faster than the Apple, less use of these routines is necessary. A routine written in Apple's machine language will not work on ADAM. If you see one, either give up on your translation or use an Applesoft reference to find out what the routine does. Then you can translate *what it does* into SmartBASIC or Z80 machine language. A good reference for the machine languages routines in Applesoft is Lon Poole's *Apple II User's Guide* published by Osborne/McGraw-Hill, and a good reference for Z80 machine language is Alan Miller's *8080/Z80 Assembly Language* published by John Wiley and Sons.

Files

Here the major weaknesses of ADAM's data packs and operating system (in Release 1.0 of SmartBASIC) show up. As mentioned in Chapter 16, in SmartBASIC the OPEN command does not set the file pointer. In Applesoft, it does. If you suspend INPUTting from or PRINTing to a file, you must restart either from the beginning or end. This is completely incompatible with the Apple. If an Applesoft program has

1. two files open at the same time
2. the command PRINT *d*$ (followed by nothing)

you will have a hard time translating the program into SmartBASIC. Unless all the open files can fit into memory (RAM) at the same time, certain features of SmartBASIC make it impossible to translate such programs from Applesoft.

Applesoft allows you to increase the number of file buffers up to 16 using the MAXFILES command. ADAM has only two buffers.

Applesoft also allows an ASCII file to control the action of the Apple. The command is EXEC. SmartBASIC has no similar command.

GLOSSARY

(The number in parentheses show the first chapter in which the term was used.)

ACCESS (16) — To read some stored information or otherwise get hold or control of something.
ALIGN (11) — To line up, as with entries in a table.
ALPHANUMERIC (4) — An adjective meaning a combination of letters, numbers, spaces or other typewriter characters.
ARRAYS (11) — A collection of variables each of which can be identified by:

1. the name of the array
2. one or more subscripts indicating its position in the array.

ARTIFACT (13) — Something, often a command, that has survived from an older programming language. The term is not complementary.
ASCII CODE (8) — ASCII is are an abbreviation for American Standard Code for Information Interchange. The numbers used to represent characters on most computers are written in ASCII code.
ASSEMBLY LANGUAGE (2) — A language used to communicate with a computer. It is very close to *machine language*.
ASSIGNMENT (4) — A type of BASIC instruction consisting of two parts separated by an equal (=) sign. The left-hand side is the variable name. The right-hand side holds the value given to the variable.
BACKUP (7) — A copy of a program or data file kept in case the original becomes damaged or lost.
BASIC (1) — The initials of *Beginner's All-purpose Symbolic Instruction Code*. The programming language in widest use. SmartBASIC is one version of BASIC.
BINARY SEARCH (10) — A fast method of searching in an ordered list. The list is continually divided into two parts and only the half that must contain the entry sought is then checked.
BLEEDING (14) — The oozing of distinct colors over one another as in certain color graphics displays.
BODY of a LOOP (6) — The commands between the beginning of a loop and the command to return to the beginning of the loop. For example, the commands lying between the commands FOR and NEXT.
BOMB (4) — When a program rolls over and dies.
BRANCH(ING) (5) — A place in a program where the next instruction is not necessarily the next processed.

BREAK(ING) (5) — Forcing a computer to stop processing instructions. In SmartBASIC this is done with CONTROL/C.
BUFFER (16) — A space in memory that temporarily holds information going to or coming from the computer.
BUILT-IN-FUNCTION (9) — A function that exists in a programming language like SmartBASIC.
CALCULATOR MODE (3) — In this mode of operation, a computer executes instructions as they are entered. Also called *direct* or *immediate mode.*
CHARACTER (3) — Usually a letter or numeral, but may also be a special symbol like punctuation marks or even a space.
CHARACTER SET (8) — A set of symbols; usually refers to those built into a computer.
CHIP (2) — An informal term for an *integrated circuit.* So-called because integrated circuits sit on little chips or pieces of silicon (sand).
CODING (INTERLUDE 1) — Writing instructions for ADAM in some language it can understand or translate.
COMPILER (2) — A program that translates another program into ADAM's machine language. Unlike an *interpreter,* all statements are translated before any are executed.
COMPUTERESE (1) — The (sometimes strange) vocabulary used by people involved in computers.
CONCATENATION (13) — The joining together (technically, the juxtaposition) of words or strings.
COUNTER VARIABLE (6) — A variable used in a FOR-NEXT loop to keep count.
CURSOR (3) — The flashing line that acts as a placemarker on the display screen.
DATA (2) — Information used by a computer.
DEBUGGING (4) — Getting a program to run correctly.
DECREMENTING (5) — To count down using a computer.
DEFAULT SHAPE (15) — The one *shape* built into ADAM's memory.
DEFAULT VALUE (4) — A value automatically assigned unless you specify otherwise.
DELIMITER (10) — One or a combination of characters used to indicate the beginning or end of something else. For example, a number or string that signals the end of a DATA list.
DIGIT (13) — An individual symbol in a number. Also called a *numeral.* The word originally means finger, which is a pre-microcomputer method of counting.
DIRECT MODE (3) — See *calculator mode.*
DISABLE (6) — To shut off a device like ADAM's printer or to make some thing inoperable. An example of the second use of the word is *ONERR GOTO which disables ADAM's internal error trap.*
DOCUMENTATION (12) — The explanations and operating instructions accompanying, or contained in, a program.
DOWN (INTERLUDE 2) — The time when a computer can't work because repairs or tests are needed. Its opposite is UP.
DRIVER (Afterword) — A program, subroutine or piece of machinery that controls or operates something else.
DYNAMIC DIMENSIONING (10) — Using a variable or a calculation to dimension a list or array.
EMPTY WORD (10) — A word with no letters; the analog for strings of the number zero.
ENDLESS LOOP (5) — same as INFINITE LOOP.
ENTER (3) — Information or instructions are *entered* into a computer by hitting the return key after typing them. The return key tells the computer that the typed statements are complete and should be either acted upon or stored.
EXCHANGE SORT (10) — A *sorting* method shown in Chapter 10.
EXECUTED (5) — Those operations already performed by a computer.
EXECUTING (2) — Carrying out of program instructions by a computer.
EXPANSION INTERFACE (1) — A place on a computer where you can add on more things. (See *peripheral devices.*)
EXPONENTIAL NOTATION (3) — A very convenient method for representing extremely large and extremely small numbers.

FILE (16) — A collection of related data or a program stored on a data pack, floppy disk, computer tape or other storage medium.

FILE POINTER (16) — Usually, a memory location holding the position in a file from which a computer will retrieve or to which it will send the next piece of information.

FLAG (13) — A piece of information used to indicate the condition of a DATA list or file, peripheral device or the computer itself.

FLOATING POINT VARIABLE (11) — A variable (or box) that can be filled with decimal numbers. In ADAM these numbers range from about −1.7E38 to 1.7E38 and have nine *significant figures*.

FONT (1) — One of the many different styles used to print characters.

FOR-NEXT LOOP (6) — A loop in a program starting with FOR and ending with NEXT.

FORCE A LINE FEED (1) — A computer instruction, either programmed or built-in, that tells the printer or screen display to go to the next line.

FORMAT (verb) (4) — Set up a display or printed output so that it is appealing and easily understood.

FORMATTED PRINT (6) — A screen display or printed output that has been laid out for visual appeal and usefulness.

FUNCTION (9) — Informally, a command that takes something you give ADAM and then does something to (or with) it.

GIGO (3) — These initials stand for garbage in, garbage out, meaning that if you type something that the computer misunderstands, you get out nonsense. It is pronounced gee-go (both g's are hard).

GLOBAL VARIABLE (12) — Variables whose values are passed between subroutines.

GRAPHICS (7) — Displaying pictures. Contrasted with *text*.

HANDS-ON EXPERIENCE (1) — Actually working with a computer, rather than reading or talking about it. Sometimes this is called *getting your hands dirty*.

HANG (13) — A verb describing a computer that won't respond to commands or breaks down. The computer may be trapped in a loop or have become confused by instructions it couldn't handle.

HARD COPY (6) — A paper copy.

HARDWARE FAILURE (16) — A breakdown of a computer or peripheral other than those caused by programming.

HIGH LEVEL LANGUAGE (2) — A language, like BASIC or FORTRAN, that requires extensive translation before reaching machine languages.

HIGH RESOLUTION (7) — High grade graphics using relatively small blocks of color.

I/O ERROR (4) — An error in INPUTting or OUTPUTting information, specifically to or from the data pack drives.

IDENTIFIER (11) — A special symbol used as a label.

IMMEDIATE MODE (3) — See *calculator mode*.

IMPLEMENT (5) — To go from the statement of a problem to a program or device that solves it.

INCREMENT (5) — To count using a counter that gets progressively larger. The opposite is DECREMENT.

INDEX (11) — A number or variable that serves as an indicator of a count.

INFINITE LOOP (5) — When a computer keeps circling forever through the same instructions in a program.

INITIALIZE (6) — Assigning a definite value to a variable before the start of an operation using that variable.

INPUT (8) — Information that goes into a computer.

INTEGER (9) — A number without anything after the decimal point.

INTEGER VARIABLE (11) — A variable that can only be filled with whole numbers. In SmartBASIC they can range from −32767 to 32767.

INTEGRATED CIRCUITS (2) — Electronic devices containing several to hundreds of thousands of simpler devices like transistors. The most important integrated circuit first in a microcomputer is the *central processing unit*. ADAM's CPU is the Z-80A integrated circuit manufactured by Zilog.

PHYSICAL LINE (4) — A line on your screen. On ADAM these have 31 characters. Also called a *screen line.*

PICTURE ELEMENT (14) — A single dot on a video display screen. Also called a *pixel.*

PIXEL (4) — An abbreviation of *picture element.*

PORTABLE LANGUAGE (2) — A language that can be used on many computers.

PORTS (2) — The entrances to - and exits from - a computer.

PRETTYPRINTING (8) — Doing fancy or unusual things on the screen or printer.

PRINT ZONES (4) — The default printing areas. The area on a display or printer line that ADAM automatically uses when told to print something (unless instructed otherwise).

PRINTHEAD (8) — The part of a printer that actually prints.

PROCESSING (4) — The carrying out by the CPU of the instructions you gave to ADAM.

PROMPT (3) — A symbol (in SmartBASIC the]) that tells you a computer is waiting to take your orders.

PSEUDO-CODE (7) — Statements written in English which say what a program is supposed to do without using the precise form of a programming language.

PSEUDO-RANDOM (9) — A number that has many of the properties of a RANDOM number but is constructed using a fixed method.

QWERTY (1) — The normal layout of the typewriter keys. The layout is named after the keys above the rest position of the fingers on the left hand.

RADIAN MEASURE (9) — A measure of angle size in which 2π is one complete revolution (360°) and all angle sizes are proportional to the degree measure.

RAM (Random Access Memory) (2) — The part of a computer's temporary memory that can be changed. Anything in RAM is lost when the computer is turned off.

RANDOM (9) — By chance.

READ FILE (16) — A file that is *open* for computer to read.

REAL VARIABLE (11) — Same as *floating point number.*

RECURSION (12) — A subroutine that *calls* itself.

REFERENCE POINT (11) — A point, usually on a graph, whose location is used to specify the location of all other points.

REFERENCE VALUE (11) — A number or value on which other information is based.

RELATIONAL OPERATORS (5) — The symbols used for tests in the IF clause of an IF-THEN. They are <, >, <>, =, <=, >=.

RELATIVE COORDINATES (11) — Coordinates of a point using an origin other than (0,0).

RELATIVE SCALING (11) — Scaling of data values where only the difference of values from a *reference value* is scaled.

REPRESENT (11) — To use one type of object to stand for another, as in representing the GR screen by a 40 by 40 array.

RESERVED WORDS (16) — Strings that are used by a computer for special purposes. They generally cannot be used as variable or file names.

REVERSE VIDEO (8) — Same as *inverse video.*

RIPPLE SORT (10) — Same as *exchange sort.*

ROM (Read Only Memory) (2) — The part of a computer's internal memory that cannot be changed. It is not affected when the power is turned off.

ROUND-OFF ERROR (9) — The difference between a number and a computer approximation to that number. It is caused by computers only using a fixed number of digits (*significant figures*) in computations.

RUN (10) — To start the processing of a program.

RUNNING (4) — Another word for *executing.*

SCALE FACTOR (11) — A number which is used to multiply all relevant data or sizes of objects, so that they are increased or reduced proportionately.

SCALING (9) — Modify numbers by multiplying them by the same number.

SCIENTIFIC NOTATION (3) — See *exponential notation.*
SCROLLING (3) — When the top line on your screen disappears and everything else that's displayed on your screen moves up one line.
SEED (9) — A number used to produce a different PSEUDO-RANDOM number list.
SHAPE (15) — A set of special instructions for drawing something.
SHAPE TABLE (15) — A list and description of *shapes* stored in memory.
SIGNIFICANT FIGURES (3) — This is the number of digits that are used in computer calculations. The remaining digits are lost (replaced by zeroes).
SIMULATE (9) — To imitate using a model.
SORT (10) — To rearrange a list in a prespecified order.
SOURCE CODE (Afterword) — Computer instructions written in a language not understood by the computer. It must be translated before it can be executed. Compare *object code.*
STACK (12) — An area of RAM set aside for use by RETURN commands. It holds the place in the program that control RETURNs to. More generally, a stack is a list in memory having the property that the last entry in is the first entry out.
STARTING and TEST VALUES (6) — The beginning and end values for a *counter variable.*
STATEMENT (4) — A command or group of commands in SmartBASIC.
STORE (3) — To place something into the computer's memory.
STRING (4) — A group of alphanumeric characters joined together.
STRING ARRAY (11) — An ordered collection of alphanumeric characters treated as a whole.
STRING VARIABLE (11) — A variable (or box) that ADAM can fill with a string having no more than 255 characters.
SUB-STRING (13) — A string that's part of a bigger string.
SUBROUTINE CALL (12) — The use of the command GOSUB in a program.
SUBSCRIPT (10) — The number in parentheses used to identify the position of variables in *lists* and *arrays.*
SUPPORT (9) — The commands, functions, languages, etc. available to a computer.
SYNTAX (3) — The arrangement of words, symbols and commands that a computer recognizes.
TERMINATOR (5) — A piece of information signalling the end of a program or process.
TEST MODULE (INTERLUDE 2) — A special module used to test a program or part of a program.
TEXT (1 and 7) —
1. Characters, for example, letters or numbers.
2. Displaying words or symbols. Contrasted with *graphics.*

TEXT WINDOW (7) — In *graphics mode* (GR or HGR), the remaining four lines on your screen.
TIMING LOOP(6) — A loop whose only function is to create a pause of a specified time.
TOP-DOWN PROGRAMMING (12) — An extension of *modular programming* in which a program is built like a pyramid. Requests for action pass down. Results of operations move up.
TRAP (5) — A place in a program to send control if the program receives something it can't handle.
TURNKEY PROGRAM (12) — A program that starts running as soon as a computer is turned on.
UP (INTERLUDE 2) — The time when a computer can work. Its opposite is DOWN.
USER DEFINED FUNCTION (9) — A function not built into Smart- BASIC but which can be programmed using its commands.
USER FRIENDLY (5) — A type of program that doesn't need an expert to run it.
VARIABLE (3) — A place to hold (or store) numbers or strings. It can be thought of as the name of a box.
WRAP-AROUND (13) — A form of printing in which, when the end of a line is reached, printing continues on the next line.
WRITE FILE (16) — A file that is *open* for a computer to send information to.

Reference Guide

Notations and Format Conventions

[ ] Brackets indicate that the enclosed items are optional.

{ } Braces indicate that a choice of one of the enclosed items is to be made.

... Ellipses indicate that the preceding item may be repeated.

Italics Italics indicate generic terms. The programmer must supply the actual value or wording required.

Line Number A line number is necessary for all BASIC language statements in programming mode.

Punctuation All punctuation characters, including commas, semicolons, colons, quotation marks and parentheses, must appear as indicated.

UPPER CASE Upper case letters and words must appear exactly as indicated.

Arg Argument.

B# A byte number.

col Low resolution graphics column number. 0<=numeric expression<=39.

colh High resolution graphics column number. 0<=numeric expression<=279.

const Any numeric or string constant.

D# Drive number. The default drive specification is D1 which is the left data pack.

expr Any valid SmartBASIC expression. An expression is a formula that ADAM can evaluate as a number or string.

expr$ Any valid SmartBASIC string constant, variable or expression.

exprnm Any numeric constant, variable or expression.

filename A data pack or disk filename.

format Format of a SmartBASIC language statement.

length Integer ranging from 0 to 32767 (decimal).

line A SmartBASIC program line number.

line i The i – th line number in a program.

L# Length in bytes. Integer ranging from 0 to 32767 decimal.

memaddr The memory address referenced by a numeric expression, variable or constant. In decimal the range must satisfy −65535 <=range <= 65535. (A negative address − memaddr is the same location in memory as 65535 − memaddr.)

memloc The memory location specified by an integer constant. In decimal the range must satisfy −65535<=range<=65535.

msg The message, delimited by quotation marks, which forms a string.

nvar Numeric variable.

row Low resolution graphics row number. 0 <=exprnm <= 39.

rowh High resolution graphics row number. 0 <= numeric expression <= 158 in HGR. 0 <= numeric expression <= 191 in HGR2.

R# Specified entry (or record) in a file.

S# Slot number for input or output. 0<=n<=7.

sub Subscript.

var Numeric, integer or string variable in SmartBASIC.

varnm A numeric variable name.

var(sub) Any subscripted integer, numeric or string variable in SmartBASIC.

V# A data pack (or disk) volume number. 0<=n<=255. The volume number is ignored in the minimal configuration.

Arithmetic and Relational Operators

ARITHMETIC

addition	+	
multiplication	*	
exponentiation		

RELATIONAL

equal	=	
not equal	<>	(optional ><)
less than	<	
greater than	>	
less than or equal	<=	(optional =<)
greater than or equal	>=	(optional =>)
subtraction and unary minus	−	
division	/	

BOOLEAN

negation (logical compliment)	NOT
and (logical addition)	AND
or (logical or, not exclusive)	OR

Notes:
(1) + is also used for concatenation (joining) of strings.

ON varn GOSUB line [,line [, . . .]] Is a conditional GOSUB. If the truncated value of varn is i, control is passed to the subroutine at the i'th line listed. If i is outside the range, control passes to next numbered line.

ON GOTO Is a conditional GOTO, works exactly like ON GOSUB.

POP If subroutine B is called by subroutine A and B contains a POP, then subroutine B returns to process the statement following the call to subroutine A.

RETURN used with a GOSUB, sends processing back to the statement after the GOSUB.
format: RETURN

Error Trapping

ERRNUM( ) Function which returns the error code of an error.

CLRERR Cancels the preceding ONERR GOTO.

NOTRACE Turns off TRACE.

ONERR GOTO line Forces program processing to the line number indicated if an error is found. Must precede the error in a program.

RESUME Sends program processing to the statement where the error occurred.

TRACE When activated displays the line number of each statement as it is processed.

Machine Language Interface

CALL memaddr Branches to a machine language subroutine.

HIMEM :exprnm sets the highest location in RAM available for SmartBASIC.

LOMEM :exprnm sets the lowest memory location available for SmartBASIC.

PEEK(memaddr) Returns the decimal value of the contents of a memory location specified in decimal.

POKE memaddr,exprnm Places the value of a numeric expression (between 0 and 255) in a specified memory location. (Both must be expressed in decimal.)

USR(exprnm) Stands for User Supplied Routine. Causes ADAM to process a machine language function.

WAIT exprnm,exprnm [,exprnm] Suspends processing until a specified value occurs at a specified port.

Processing Statements

DATA const,const, ... Sets up a list of values to be assigned to the variables from a READ statement. These values can be either string or

DEF FN var(arg) where arg is a dummy variable. Allows a user-defined one variable function.

DIM var (exprnm[,exprnm[, ...)[,]]var(exprnm, . . .)] Sets up space in memory for an array. The default value is 10. All subscripts must be positive. Always includes a 0'th element.

END Terminates program execution.

FOR varn = exprnm TO exprnm [STEP exprnm]
1 2

Exprnm is automatically increased by the value 1 (or the 1 STEP value) each time a NEXT is encountered. When the varn exceeds exprnm control passes to the line after the 2 NEXT.

[LET] var = expr Assigns a value to a variable.

NEXT [varn] Causes the counter for the preceeding FOR statement to be increased by the STEP value (or 1 if no step is given). Note: varn is optional and eliminating it speeds up processing slightly.

READ var[, var[, var, . . .]] Assigns values from a DATA statement sequentially.

REM [msg] Non-executable statement used for remarks.

RESTORE Resets the data pointer to the first entry in the first DATA statement.
format: RESTORE

STOP Stops program execution. Also CLOSEs all open files.

Input/Output Controls

GET var Accepts one character from the keyboard without displaying a ? or the character on the screen. (Caution: overrides the CONROL/C interrupt.)

IN# Activates an input device for subsequent input. The constant must be between 0 and 7.

INPUT [" prompt "]; var [var,] Will accept data to be assigned to a variable. Optionally can display a prompt. The input comes from the keyboard unless some other device (or read file) is activated. In that case, the input comes from the device (or file).

PR# Selects the device that will receive output. (Caution: remains in the previous state unless disabled by another PR# command.)
PR#0 Is the television (or monitor).
PR#1 Is the SmartWRITER printer.

PRINT Sends output to the device (or file) previously specified.

PRINT expr[,expr[, . . .]] Causes the different expressions to be printed in separate print zones.

PRINT expr[;expr[; . . .]] Causes the different expressions to be printed adjacent to one another.

PRINT SPC(exprn) Can only be used in a PRINT statement. Moves the cursor or printhead the designated number of spaces to the right (0 < exprnm < 255.).

PRINT TAB(exprn) Can only be used in a PRINT statement. Moves the cursor or printhead the designated number of spaces to the right. (0 < exprnm < 255.).

All digital data pack commands when used in a program use **PRINT CHR$(4);"command"**

APPEND Opens and allows the addition of data to the end of an already existing sequential file.
format: PRINT CHR$(4); **"APPEND filename"**

BLOAD Loads a binary file into memory at the location specified.
format: BLOAD filename , A# [,D#] where A# is the memory address and D# is the optional device #

b**BRUN** Loads and runs a binary program.
format: BRUN filename A# [,D#]. The A# and D# have preset default values that can be changed.

The screen displays 31 characters in SmartBASIC.

COLORS

	LOW RESOLUTION	HIGH RESOLUTION
0	BLACK	BLACK - 1
1	MAGENTA	GREEN
2	DARK BLUE	VIOLET
3	DARK RED	WHITE - 1
4	DARK GREEN	BLACK - 2
5	GREY - 1	ORANGE
6	MEDIUM GREEN	BLUE
7	LIGHT BLUE	WHITE - 2
8	LIGHT YELLOW	BROWN
9	MEDIUM RED	DARK BLUE
10	GREY - 2	GREY
11	LIGHT RED	PINK
12	LIGHT GREEN	DARK GREEN
13	LIGHT YELLOW	YELLOW
14	CYAN (SKY BLUE)	AQUA
15	WHITE	MAGENTA

COLOR = exprnm Sets the low-resolution graphics color.

DRAW exprnm AT colh, rowh Draws a high-resolution graphics shape on the video display.

FLASH Causes the video display to alternate between light on dark and dark on light. Only works in TEXT mode.

GR Puts the display into low-resolution graphics mode with 40 col and 40 rows.

HCOLOR = exprnm Sets the color for plotting in the high-resolution graphics mode.

HGR Puts the video display into the high-resolution graphics mode of 256 columns x 160 rows, with a four-line text window.

HGR2 Sets the display to a full-screen high-resolution graphics mode of 256 columns x 192 rows.

HLIN col #, col # AT row # Draws a horizontal line on the display from col to col at the specified row in low-resolution graphics.

HOME Clears the display and positions the cursor at row 1, column 1 or at the beginning of the text window.

HPLOT colh, row Places a dot or draws a line on the display in the high-resolution graphics mode.

other formats
HPLOT TO colh, rowh
HPLOT colh#,rowh# TO colh#,rowh#
[TO colh#,rowh#, . . .]

HTAB col or HTAB(col) Moves the cursor to the specified column.

INVERSE Puts display output into the inverse video mode.

NORMAL Turns off FLASH and INVERSE.

PLOT col#, row# Displays a block on the low-resolution graphics display.

POS(exprnm) Returns the column position of the cursor.

ROT = exprnm Rotates high-resolution shapes to be drawn by DRAW or XDRAW. exprnm must be between 0 and 255. One full rotation is 64. The ROT command repeats after 64; ROT = 80 is the same as ROT = 16 (a quarter turn).

SCALE = exprnm Sets the size of high-resolution shapes to be drawn by DRAW or XDRAW. exprnm must be between 0 and 255. Cautions: SCALE must be set before using DRAW or XDRAW. 1 not 0 is the smallest scale (0 is the largest).

SCRN(col,row) Returns the color code of the low-resolution graphics point at the specified coordinates.

SPC(exprnm) Function that moves the cursor a specified number of positions to the right.

SPEED = exprnm Changes the rate at which characters are output to the display. exprnm must be between 0 and 255. Caution: remains at last speed set. Default is 255.

TEXT Returns display to the full-screen mode from any graphics mode.

VLIN row#, row# AT col# Draws a vertical line from row to row in column col on the display in the low-resolution graphics mode.

VPOS(exprnm) Returns the vertical position of the cursor. The first line is number 0, the last line is number 23.

VTAB row or VTAB(row) Positions the cursor to the line specified by row in the current display column.

XDRAW exprnm [AT colh, rowh] Draws shape exprnm at the designated location in the complementary color. Only works after a previous DRAW command. The effect is to erase the shape.

Basic Functions

ABS(exprnm) Returns the absolute value of a number.

ASC(expr$) Returns the ASCII code number for the specified character.

ATN(exprnm) Returns the arctangent of an angle of exprnm radians.

CHR$(exprnm) Returns the character (string value) of the specified ASCII code.

COS(exprnm) Returns the cosine of an angle of exprnm radians.

EXP(exprnm) Returns the base of the natural logarithm (e) raised to the specified power.

FN varnm(exprnm) Calls the previously defined function using the value given by exprnm.

FRE(exprnm) Returns the number of bytes of memory available for an ADAM program. The value or exprnm is irrelevant but must be present. 0 is usually used.

INT(exprnm) Returns the integer portion of a number. Caution: in version 1.0 of SmartBASIC, INT can behave strangely at times.

LEFT$(expr$,exprnm) Returns the leftmost exprnm characters of the string expr$.

LEN(expr$) Returns the length of the specified string.

LOG(exprnm) Returns the natural logarithm of the specified number.

MID$(expr$#,exprnm#,[exprnm#]) Returns exprnm characters from expr$, commencing with character exprnm.

POS(exprnm) Returns the column position of the cursor. The value of exprnm is irrelevant but it must be present.

RIGHTS(expr$,exprnm) Returns the rightmost exprnm characters of expr$.

RND(exprnm) Returns a random number. Reseeds when exprnm < 0.

SCRN(col,row) Returns the color code of the specified coordinates when in low-resolution graphics mode.

SGN(exprnm) Returns +1 if exprnm is positive, −1 if negative, and 0 if its value is zero.

SIN(exprnm) Returns the sine of an angle of exprnm radians.

SQR(exprnm) Returns the square root. exprnm >= 0.

STR$(exprnm) Converts a numeric value to a string.

TAN(exprnm) Returns the tangent of an angle of exprnm radians.

VAL(expr$) Returns the numeric value of a string.

Index

ABS function, 220
accessories, 22-23
ADAMnet, 8
alphabetizing, 249
AND command, 113
APPEND command, 411, 430, 442
Applesoft BASIC, 36
arithmetic operations, 48
arrays, 255-280
  integer, 264
  numeric, 257
  string, 257
arrow keys, 80
  CONTROL and, 86
ASC command, 338, 348
ASCII codes, 198-200
assembly language, 33
assignment statements, 66
ATN function, 222
AUX VIDEO socket, 14
available memory, 264

backspacing, 202
bar graph program, 168
BASIC. *See also* SmartBASIC *and* specific commands
  Applesoft, 36
  development of, 34
binary files, 438
binary search, 247
BLOAD command, 442
block diagram of ADAM, 29
boldface program, 328
branching, 90
BRUN command, 442
BSAVE command, 443
buffer, 412
built-in functions, 220-224
byte, defined, 32

calculator functions, 40
caret key, 43
cassette, data pack, 10, 11, 407
  care of, 17
CATALOG command, 73, 77, 443
central processing unit (CPU), 30
character, defined, 51
chips, 30
  layout of, 31
CHR$ command, 197, 204
circle, drawing of, 393
CLEAR command, 240, 252
CLOSE command, 411, 443
coded data files, 423
colon, 65
color
  in high-res graphics, 352
  in low-res graphics, 157
COLOR command, 156, 179, 351
commands. *See* name of command
commas, 59
compiler, 33, 35
computer insurance, 17
concatenation, 314
CONT command, 248
control keys and codes, 201-202
CONTROL/C, 92
CONTROL/L, 83
CONTROL/N, 83
CONTROL/O, 82
CONTROL/P, 64, 77
CONTROL/S, 133
CONTROL/X, 83
COS function, 222
CP/M, 23
CPU (central processing unit), 30
cursor, defined, 40
cursor control keys, 80
curves, drawing of, 387

daisy wheel, replacing, 4
data files
  adding information to, 429
  binary, 438
  coded, 423
  commands for handling, 441
  programming and, 417
  random access, 438
  use of, 434
  writing and reading of, 410
data pack, 10, 11
  care of, 17
  information organized on, 407
data pack drive, 10
  care of, 17
DATA statements, 242-246, 252, 258
debugging, 75, 305-311
DEF FN, 225
DELETE command, 409, 443
delimiter, defined, 245
digital data pack drive, 9
  care of, 17
DIM, 234, 252
dimension statement, 234
direct mode, 41
disk drive, floppy, 22
dollar sign ($), 63, 67
DRAW command, 384, 404
drawing with paddles, 374
drive, digital data pack, 9
  care of, 17
dynamic dimensioning, 237

E notation, 44
editing, line, 80-88
END statement, 59, 77
equal () operator, 103
ERRNUM(1) command, 417
ERROR message
  I/O (input/output), 72
  SYNTAX, 42
error trap, 291, 422, 426
execute, defined, 28
EXP function, 221
expansion interface, 9
exponentiation, 43

fanfold paper, 5
files, data, writing and reading of, 410
flag, 321
FLASH command, 190, 204
floppy disk drive, 22
FN, 225
format, defined, 70
FOR-NEXT loop, 126, 151
  nested, 145
FORTRAN, 33
FRE command, 264

functions. *See also* name of function
  built-in, 220-224
  calculator, 40
  trigonometric, 221
  user-defined, 224, 296

game paddles, 8
GET command, 341, 348
GOSUB command, 283, 304
GOTO, 90
GR command, 154, 179, 351
graphics, 12
  commands for, 351
  high-res. *See* high-res graphics
  low-res, 157, 171-178, 266
  paddles and, 370-379
graphics mode, 154
graphing program, 268-275, 395-397
greater than (>) operator, 103
grid-point, 274
grounding, 15

hard copy, 141
HCOLOR command, 351, 379
heads, drive, care of, 17
HELLO program, 287
HGR command, 351, 358, 379
HGR2, 365, 379
hierarchy of operations, 46
high-resolution colors, 352
high-resolution graphics, 349-404.
  See also names of commands
  colors in, 352
  curves in, 387
  drawing in, 374
  paddles and, 370-379
  shape manipulation in, 382
  text window in, 365
HIMEM command, 451
HLIN command, 159, 179
HOME command, 41
housekeeping, 414
HPLOT command, 351, 359, 379
HPLOT TO, 360, 379
HTAB command, 193, 204

IF-THEN, 102, 107
  subroutines and, 286
immediate mode, 41
IN # command, 475
INIT command, 435, 443
initialize, 130, 239
input prompt, 71
INPUT statement, 59, 77
input/output error, 72
insurance, computer, 17
INT command, 211, 215
integer, 211

integer arrays, 264
integer variable, 262
integrated circuits, 30
interference with television reception, 16
interpreter, 35
I/O ERROR message, 72
INVERSE command, 190, 204

joystick, 8

keyboard, 6-8, 7f
kilobyte, defined, 32

languages, computer, 33
LEFT$, 336, 348
LEN command, 318, 347
less than (<) operator, 103
LET statement, 50, 66
line editing, 80-88
line numbers, 57, 100
lists, computer use of, 233-242
LIST command, 76, 77, 84
LOAD command, 73, 77, 443
LOCK command, 436, 443
LOG function, 221
LOGO, 23
LOMEM command, 459
looping, 117
loops, 93, 117
  timing, 134
low-res graphics, 157, 171-178, 266
low-res graphics editor, 171-178

machine language, 33
markers, 271
mathematical functions, subroutines and, 298
memory, available, 264
memory expander, 22
memory location, PEEK and, 449
menu-driven program, 97
MID$ command, 324, 330, 347, 348
modem, 9, 22
modular programming, 100, 175
MON command, 439
monitors, 14
motherboard, 10
music, program for, 460-468

nested FOR-NEXT loop, 145
NEW command, 60, 76
NEXT command, 127, 151
noise creation, 457
NOMON command, 440
NORMAL command, 191, 204
NOT command, 115
not equal (<>) operator, 103
notes, musical, 468
NOTRACE command, 310
numeric array, 257

office model of computer, 26-28
ON-GOSUB command, 292, 304
ON-GOTO, 97
ONERR GOTO command, 417
OPEN command, 411, 443
operating system, 410
operators, relational, 103
outlet, grounded, 15
overscan, 18, 163, 356
overwriting, 202

paddle, 8, 370-379
  control of, 371
  drawing with, 374
paintbrush program, 171-178, 266, 419
paper
  loading, 6
  printer, 5
parabola, drawing of, 391-392
parentheses in mathematical operations, 46
PDL command, 370, 379
PEEK command, 449
peripheral devices, 10
pixel, 350
PLOT command, 156, 179, 351
plus sign (+), 314
POKE command, 450
polar coordinates, 397
POP command, 302, 304
ports, 30
POS function, 321
POSITION command, 444
PR# command, 475
PR# 0 command, 142, 151
PR# 1 command, 141, 151
pretty printing, 190, 318
PRINT CHR$(4), 410
PRINT statement, 40, 58
  summary of, 65
printer, 4-6
  paper for, 5
printing, 64
program
  defined, 26
  modular, 100
  structured, 35
  top-down, 293
prompt
  defined, 40
  input, 71
pseudo-random numbers, 207, 227

quotation marks, 58, 197

radian measure, 221
RAM (random access memory), defined, 30
random access files, 438
random number generator, 206, 227
rational function graph program, 395-397


$22.95

Computers

IN CANADA
CANADA

The Whole Family's Guide to the Ultimate Family Computer

THE BASIC ADAM®

Everyone and anyone can use Coleco's Adam—and this Self-Teaching Guide can help them use it well. It's for both those just catching on to programming and for sophisticated computer jocks looking for a workout. Equipped with an expandable 80K memory, a built-in word processor, a high-speed digital memory drive, daisy wheel printer, *and* game system, Coleco's Adam is unmatched for value, ease of use, and flexibility.

For first-time users, *The BASIC Adam* assumes no prior computer experience. When you're new to computers, sometimes just unpacking the machine and setting it up can seem forbidding. This guide takes you step-by-step through your Adam's assembly—then gets you started on your first SmartBASIC statements—with less fuss and bother than you thought possible.

For advanced programmers, *The BASIC Adam* opens up a new world of graphics, filing systems, and subroutines.

It's the ultimate guide for the ultimate family computer.

William Abikoff teaches basic computer programming to prospective elementary education teachers and is a professor at the University of Connecticut. He was formerly a systems engineer and computer programmer at Bell Labs.

Gary Cornell teaches computer literacy at the University of Connecticut at Storrs.

Any questions regarding this book should be addressed to the publisher, John Wiley & Sons, and not to Coleco, which is not responsible for and was not involved in the creation of this book.

Adam® is a registered trademark of Coleco, Inc.

ISBN 0 471 80807-5

Wiley Self-Teaching Guides have taught more than two million people to use, program, and enjoy microcomputers. Look for them all at your favorite bookshop or computer store.

WILEY PRESS
a division of JOHN WILEY & SONS, Inc.
605 Third Avenue, New York, N.Y. 10158
New York • Chichester •
Brisbane • Toronto • Singapore

A Self-Teaching Guide

Call 1-800-428-3696 for the name of your nearest Sams retailer or distributor (residents in IN, AK, HI call 317-298-5566).

If your retailer or distributor doesn't stock the Sams publication you need, you can order it through him or directly from us. Orders placed directly with us are subject to a $2.00 additional handling charge per order.

PHONE ORDERS

Depending on where you live, call either number listed above and charge your Sams publication to your VISA or MASTERCARD.

MAIL ORDERS

Use the order form below, or if it's missing, send your order on a plain piece of paper.

(1) Include your name, street or RD address, city, state, and zip.

(2) Tell us the titles of the books you need, the product numbers (see other side of this card) and the quantity of each one you'd like.

(3) Add up the total cost for the books, add your state's sales tax if applicable, and then add $2.00 for handling.

(4) Include your check or money order for the full amount due, or

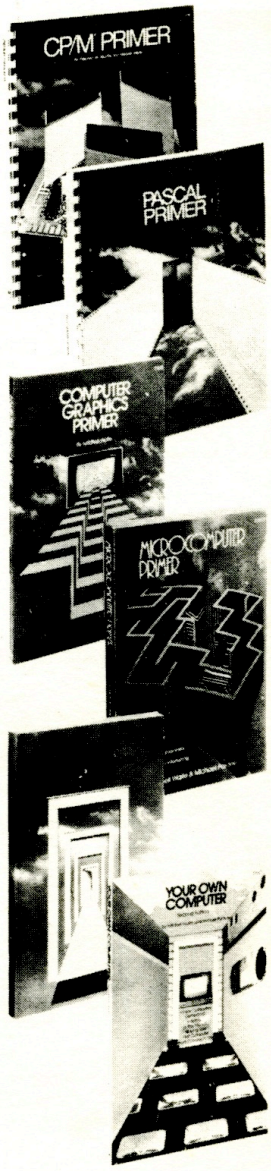
(5) Charge it to your VISA or MASTER CARD. To charge it, list your account number and the card expiration date. Shipping charges will be added to credit card orders.

(6) Mail your order to:

Howard W. Sams & Co., Inc.
Dept. No. X0438
4300 West 62nd St., P.O. Box 7092
Indianapolis, Indiana 46206

In Canada, contact Lenbrook Electronics, Markham, Ontario L3R 1H2

All books available from Sams Distributors, Bookstores, and Computer Stores. Offer good in U.S. only. Note: Distributor, Computer Store and Dealer inquiries are welcome.



See other side for list of Sams/Waite Primers and their prices.