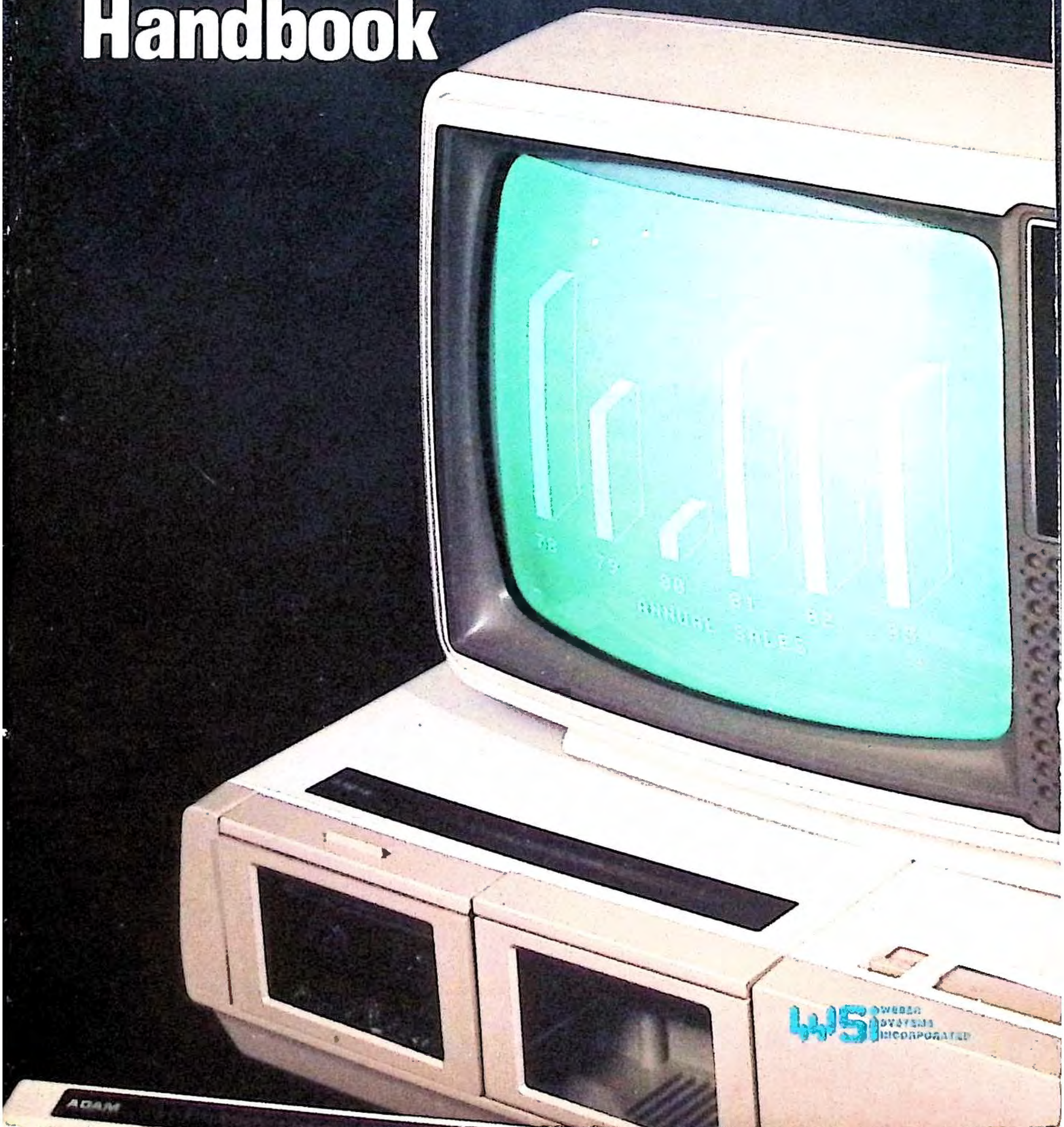
COLECO ADAM™ User's Handbook



WSI WEBER SYSTEMS INCORPORATED

Coleco ADAM™ User's Handbook

Coleco ADAM™
User's Handbook

by
WSI Staff

Weber Systems, Inc.
Cleveland, Ohio

The authors have exercised due care in the preparation of this book and the programs contained in it. The authors and the publisher make no warranties either express or implied with regard to the information and programs contained in this book. In no event shall the authors or publisher be liable for incidental or consequential damages arising out of the furnishing, performance, or any information and/or programs.

Published by:
Weber Systems, Inc.
8437 Mayfield Road
Cleveland, Ohio 44026

For information or translations and book distributors outside of the United States, please contact WSI at the above address.

Coleco ADAM™ User's Handbook

Library of Congress Cataloging in Publication Data
Main entry under title:

Coleco ADAM User's Handbook.

Includes index.
1. Adam (Computer) — Programming. 2. BASIC (Computer program language) I. Weber Systems, Inc.
QA76.8.A153c65 1984 001.64 84-2240
ISBN 0-938862-45-6

Typesetting and Layout: Tina Koran, Jana Butler, and Maria Stamoulis

Contents

Introduction

The **Coleco ADAM User's Handbook** is a complete guide to the operation of the ADAM computer system. The ADAM offers powerful word processing and programming capability, both of which are covered in depth in this book. The combination of tutorial and reference material contained in this book make it an ideal guide for beginning as well as experienced ADAM owners.

Chapter 1 provides an overview of the ADAM as well as an introduction to the concepts of computing. No prior knowledge of computing, word processing, or programming is assumed.

Chapter 2 contains a complete description of ADAM installation. A troubleshooting guide is also included to help the reader solve common equipment problems.

Chapter 3 contains a comprehensive guide to the SmartWriter word processor. This chapter includes a tutorial on SmartWriter operations, complete with examples and convenient reference material. Keyboard usage is also discussed in this chapter.

Chapters 4 through 11 provide extensive information concerning the techniques used to program the ADAM in the SmartBASIC programming language. These chapters cover all aspects of computing, including file handling and graphics.

The final chapter is a detailed reference guide to the SmartBASIC language. This chapter allows a programmer to quickly recall the exact structure of any program statement.

The **Coleco ADAM User's Handbook** combines tutorial and reference chapters with useful appendices to provide a valuable guide for all ADAM owners.

1

Introduction to the ADAM and its Peripherals

Overview

The decade of the 1980's has seen a phenomenal growth in the information processing power brought to consumers at affordable prices. Tasks which would have required huge mainframe computers only a few years ago are now commonplace, and can be undertaken by inexpensive home machines. The Coleco ADAM is just such a machine, and its combination of price, ease, and features make it one of the finest offerings yet brought to today's market.

With ADAM, you can play hundreds of games. In addition to Coleco game cartriges, you can play cartridges designed for the Atari 2600 video computer system by Activision, Imagic, Parker Brothers, and many more.

ADAM also contains an advanced word processor and letter quality printer. With these you can professionally prepare any written document from homework to a best seller.

BASIC is fast becoming a second language in our computer conscious society. With ADAM's powerful BASIC interpreter, you can automate tedious or complicated routines.

The ability to rapidily process large quantities of information is not enough: sophisticated applications programs must also be available in order to fully exploit the power of the computer. CP/M is the key to a treasure of programs which enable you to do financial analysis, record keeping, and countless other tasks. CP/M will soon be available for the ADAM.

System Components

ADAM is a complete home computer system — it includes hardware which enables the user to write letters, play games, and run state-of-the-art applications programs. Let's go for a brief tour of the equipment that performs these tasks.

SYSTEM UNIT

Figure 1.1 shows the heart of ADAM — the system unit. Contained in the system unit are several **microprocessors**, 80 kilobytes of **random access memory** (RAM), the SmartWriter word processor on a **read-only memory** chip (ROM), a Digital Data Drive, and a game slot.

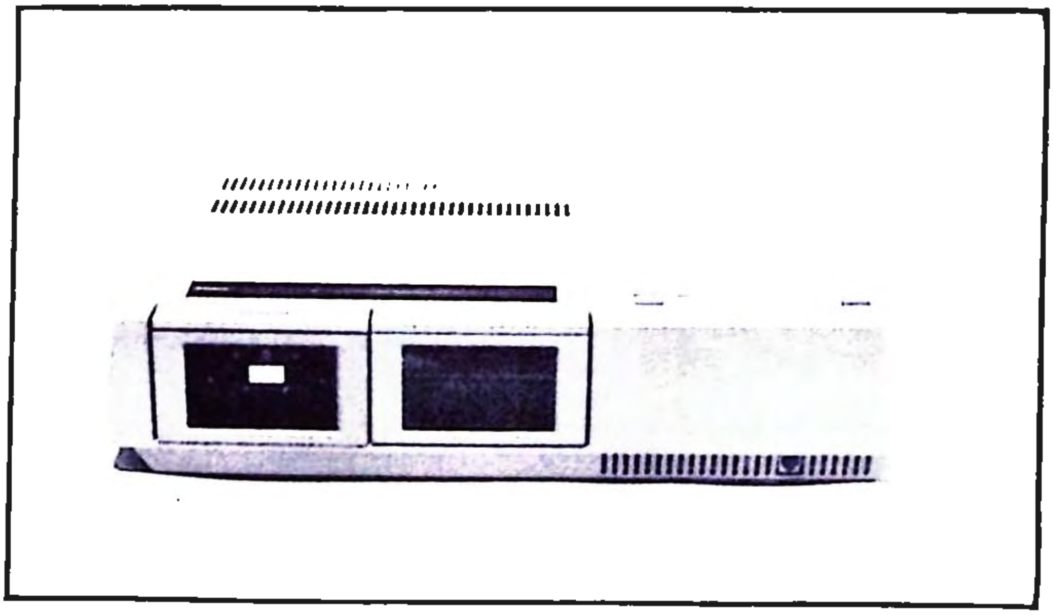


FIGURE 1-1. The ADAM System Unit

ADAM NET

The **CPU** or central processing unit is the heart of any computer. The CPU consists of the circuitry that interprets and executes the instructions issued to the computer. With most personal computers, the CPU functions are generally combined on a single chip, known as a **microprocessor.**

ADAM's CPU contains several microprocessors, each of which controls a specific action of ADAM. Because each part of ADAM has its own brain, ADAM can execute two or three tasks simultaneously. This network of **multi-tasking** is collectively known as ADAM NET.

The principal microprocessor in ADAM NET is the Z80A. This performs all the logic and math operations. ADAM NET also includes four 6801 microprocessors. Three of these control the tape drive, keyboard, and printer. The final 6801 acts as a foreman, getting instructions from the boss (Z80A), and then delegating the work.

BITS AND BYTES

ADAM's microprocessors must be given instructions to perform their work. These instructions are issued in **machine language.** All machine language instructions are given in **bits.** A bit may assume one of two values: 1 or 0. Imagine the difficulty of writing and entering correctly a string of ten thousand 0's and 1's which would tell ADAM how to play a game!

0 and 1 make up the digits of a base 2 or **binary** number system. Early in the history of computing, someone noticed that base 2 numbers could easily be converted to base 16 (**hexadecimal**). Conversion to base 16 accomplishes two things: it reduces the number of digits required to denote data and it increases the variety of available digits. Moreover, a hexadecimal string can easily be converted to its binary equivalent.

Since people communicate with symbols, mainly the alphabet and numbers, codes were devised to convert these symbols to base 2. The code that ADAM recognizes is known as **ASCII** (American Standard Code for Information Interchange). ASCII codes can assume a value from 0 to 255. In hexadecimal, 255 is represented by FF; in binary by 1111 1111.

Notice that 8 bits are required to represent an ordinary ASCII symbol. This grouping of 8 bits is so common that it has come to have a name of its own, a **byte.** A group of 1024 bytes also has a name of its own, a **kilobyte,** this is often used as an abbreviation for kilobyte. For example, 10K is shorthand for 10 kilobytes.

The Z80A is an 8-bit microprocessor. The Z80A is capable of working with 65,535 memory locations or **addresses.** In other words, Z80A can directly address 64K of memory.

ADAM includes 80K of main memory. 64K of that memory is directly addressed by the Z80A processor. The remaining 16K of RAM is reserved for ADAM's graphics capabilities.

ROM AND RAM

The CPU must call upon memory chips in order to store and retrieve information. Two kinds of memory chips have come into common usage: random access and read only memory.

Random Access Memory, RAM, is used like a large blackboard by the CPU. A memory cell or address may have information written to it or read from it. The contents of an address may also be overwritten with new information. When ADAM is powered off, any information stored in RAM will be lost.

Read Only Memory (ROM), on the other hand contains information which cannot be changed. The SmartWriter program is stored in ROM. The main advantage of storing a program in ROM is that it can be accessed almost immediately, instead of having to wait for it to be loaded from a Digital Data Pack. Another advantage is that a program stored on ROM cannot accidently be altered or erased.

Digital Data Pack

Each job that ADAM performs is accomplished by its CPU. The CPU obtains its instructions from ADAM's memory. If the necessary instructions are contained in ROM, the CPU merely fetches them. However, if the necessary instructions are to be found in RAM, they must first be stored there. Two of the most common ways of doing this are through the keyboard or from a Digital Data Drive.

The Digital Data Pack, shown in Figure 1.2, is similar in appearance to an audio cassette. Information is stored on the coating of the tape as regions of magnitized and unmagnitized particles. The read/write head inside the Digital Data Drive converts these magnitized and unmagnitized regions into binary code. The **operating system** then loads these instructions into RAM. An operating system can be defined as a group of programs which manage the computer's operation. The ADAM's operating system is stored in 8K of ROM.

FIGURE 1-2. Digital Data Pack

Although audio cassettes appear to be identical to Digital Data Packs, do not try to interchange them. Each has been engineered to perform a specific task.

If you own a cassette recorder or player, you will notice that the Digital Data Pack will respond more quickly than an ordinary audio cassette. The data transfer rate of a Digital Data Pack approaches that of a floppy disk.

Digital Data Packs are capable of storing up to 250K of information each. This translates to about 150 pages of double spaced text.

Digital Data Packs must be treated with caution. The tape surface can easily be damaged if the cassette is not treated carefully. Since information is recorded magnetically on the tape surface, all external magnetic fields must be avoided. Many common appliances such as television, telephones and stereo speakers radiate magnetic fields strong enough to damage information stored on a cassette. Try to keep cassettes at least a foot away from any equipment that draws a significant amount of power.

Like a stereo record, dust and scratches will also damage the cassette recording surface. The plastic case the cassette arrived in serves to protect the exposed magnetic surface when the cassette is not in use.

Humidity and heat are also enemies of an error-free cassette, so always store cassettes in a cool, dry place.

The ADAM's Digital Data Drive has a small door which must be opened to insert or remove cassettes. The door can be opened by pressing a switch located above it.

When information is read from or written to a Digital Data Pack, the Data Drive first locates the desired file. If a Digital Data Pack is removed while the drive is active, there is a good chance that data will be destoyed on the Data Pack. If it is imperative that a read or write operation be interrupted, a warm boot is the recommended procedure. A warm boot is accomplished by pulling the RESET COMPUTER switch located to the left of the game cartridge slot. Although a warm boot will not harm any data already stored on the Data Pack, it is likely that the particular file that was being saved or loaded will be destroyed. Therefore, a Data Pack read or write operation should be stopped only as a last resort.

Always insert cassettes into drives with the manufacturers label facing up, as shown in Figure 1-3. Never use excessive force when inserting the cassette.

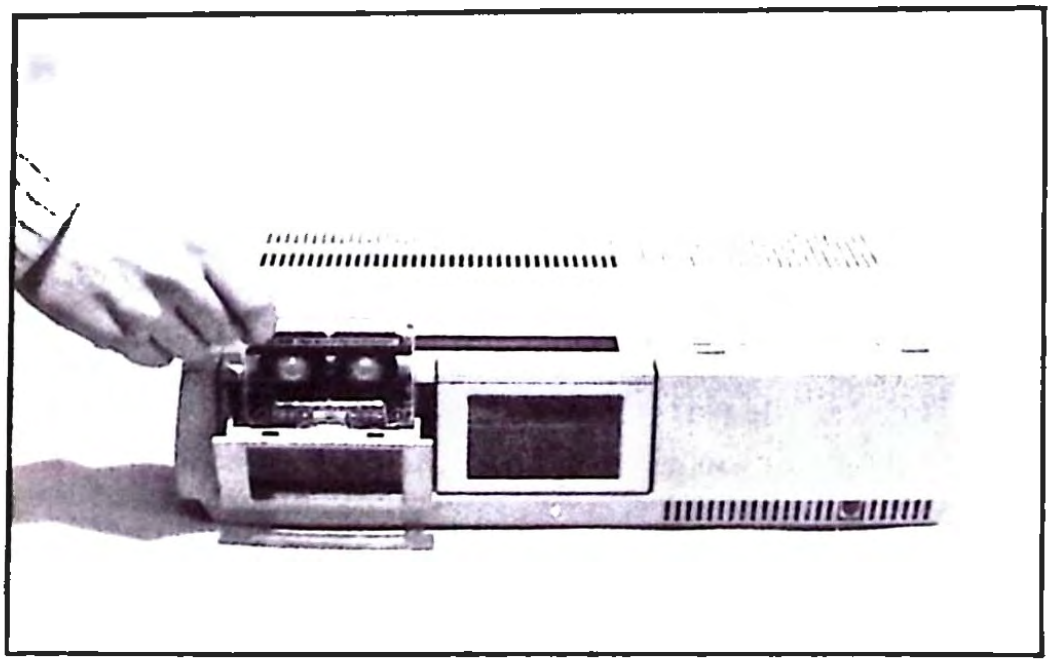


FIGURE 1-3. Digital Data Drive

When the cassette has been inserted in the drive, close the door. The door must be shut for the device to work. To remove the cassette, flip the door open and carefully pull.

KEYBOARD

ADAM comes with a professional quality keyboard with 75 full-travel keys. In addition to the standard typewriter keys, the ADAM includes special keys which allow the cursor to be moved around the display. Keys are also available which allow special instructions to be issued with a single keystroke.

LETTER QUALITY PRINTER

ADAM's letter quality daisy wheel printer allows output on almost any kind of paper. The printer is bidirectional, that is, it types from left to right, advances the carriage a line, and then types the next line from right to left. This enables it to print at a rate of 120 words per minute.

In addition to the standard pica (10 characters per inch) daisy wheel which comes with ADAM, other daisy wheels can be installed. This means that by changing the wheel, special characters such as Greek or mathematical symbols can be output. With an optional print head, available soon, 12 character per inch wheels can be installed, allowing additional characters to be output on each line.

BASIC INTERPRETER

As mentioned earlier, ADAM's Z80A central microprocessor only understands instructions consisting of sequences of binary code. A **machine language** program uses binary or hexadecimal numbers to issue instructions to the microprocessor. An assembly language program uses a phrase known as a **mneumonic** to communicate instructions. JMP is an example of an assembly language mneumonic. Machine and assembly languages are known as **low-level** languages. While execution of low level code is quite fast, programming in low level languages is a tedious, time-consuming task.

To remedy this, **high level** languages have been developed which more closely resemble English. BASIC, which stands for **Beginners All-purpose Symbolic Instruction Code,** has emerged as the most popular microcomputing language. With BASIC, the programmer need not concern himself with details like register manipulation and memory management.

If you are not already a programmer, BASIC probably should be the first language you learn. Over the years, many BASIC **applications programs** have been developed which can help solve a variety of commonly encountered problems. Once you have learned BASIC, you will be able to write programs that will solve problems that you encounter.

ADDITIONAL SOFTWARE

In addition to the SmartBASIC interpreter and SmartWriter word processor included with your ADAM, Coleco plans to introduce more software to help make the user make the most of his or her ADAM. This software should be available sometime in 1984.

The most important of these software announcements is the future availability of CP/M for the ADAM. The CP/M operating system is by far the most commonly used on 8-bit microcomputers. CP/M is the key to implementing thousands of available programs.

2

Installation and Troubleshooting

Introduction

In this chapter, we will explain in detail the steps necessary to set up the ADAM. We will also discuss many commonly encountered problems, along with the steps that should be taken to correct these problems.

Installing ADAM

Coleco has simplified the installation procedure to the point where almost anyone can set up the unit on his own. We would, however, advise that the unit be installed by an adult. Some of you purchased the ADAM family computing system, while others expanded the ColecoVision Video Game System with Expansion Module #3. Accordingly, two installation procedures will be detailed.

INSTALLING THE ADAM FAMILY COMPUTER SYSTEM

Unpack the ADAM components from their shipping carton. Check the parts against those listed on the following page.

- System Unit
- Keyboard Unit
- Daisy-Wheel Printer;
  - Pica 10 pitch daisy wheel (installed)
  - Carbon Ribbon Cartridge (installed)
- Pair of Game Controllers:
  - 1 Holder to attach controller to keyboard
- 3 Digital Data Packs:
  - SmartBASIC Digital Data Pack
  - Blank Digital Data Pack
  - BUCK ROGERS Planet of Zoom Super Game Pack
- Cords:
  - Keyboard to system unit
  - TV switch box to system unit
  - Printer to system unit
- Antenna Switch Box
- Manuals:
  - ADAM Set Up Manual
  - ADAM Word Processing Manual
  - ADAM SmartWriter Easy Reference Guide
  - ADAM SmartBASIC Manual
  - ADAM Super Game Pack Manual
- 3-to-2 Prong Plug Adapter

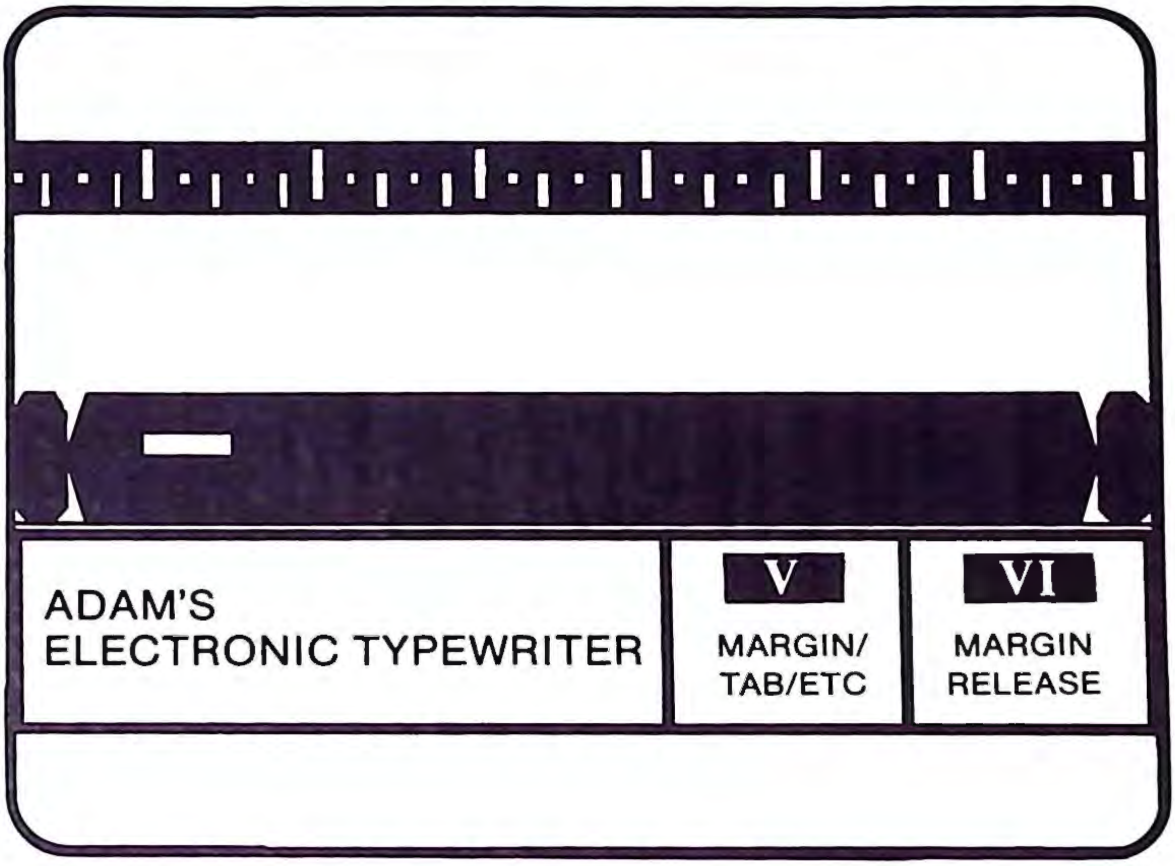
DIRECTIONS FOR SETTING UP ADAM

1. Connect the keyboard to the system unit with the cord provided.
2. Connect the joystick cords to the system unit.
3. Connect the Controller Unit to the Keyboard and attach one of the joysticks.
4. Connect the printer to the system console with the cord provided.
5. Connect the antenna switch box to the system unit.

NOTE: the TV should be "off"

6. Connect the switch box, depending on the type of antenna and antenna connector. You will have to purchase converters if either your antenna or connector are designed for coaxial cable.
7. Plug the printer into the nearest 110/120 volt wall outlet, using the 3-to-2 prong adaptor if necessary. Make sure the outlet is grounded.

8. Set the antenna switch box to "computer".
9. Turn your television to channel 3 or 4 (for best results use a channel which your area does not receive). Make sure the selector on the back of the system unit agrees with your TV.
10. Turn the daisy wheel printer on with the switch located at the back. The standard typewriter screen should appear as shown below:

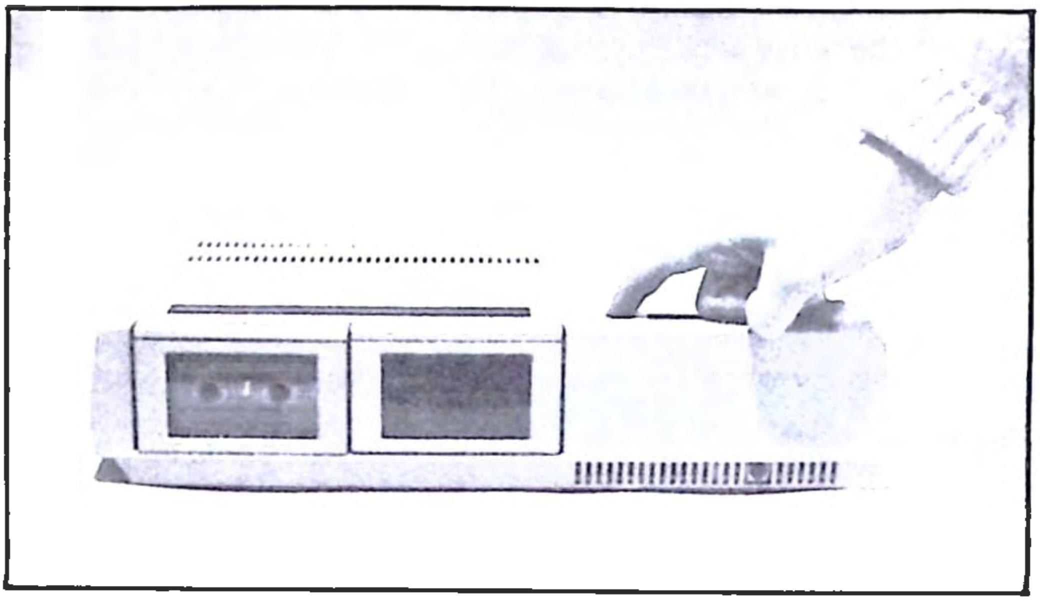


11. Insert a sheet of paper in the daisy wheel printer just as you would a typewriter. If the paper is not straight, loosen the paper release and re-align it.
12. Type the following:

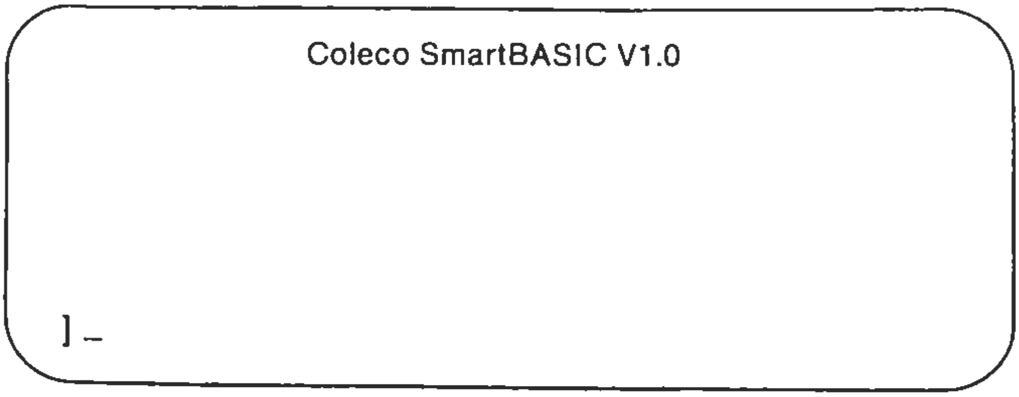
    Pack my box with twelve dozen liquor jugs

    the text should appear both on the screen and the printer.

13. Clear the screen by pulling thc RESET COMPUTER switch forward, as shown below:



14. Insert the SmartBASIC Digital Data Pack into the Digital Data Drive and pull the RESET COMPUTER switch again. The Digital Data Drive should activate and load the program. This procedure takes approximately 30 seconds. **Do not interrupt the loading procedure.** Doing so may causc damage to the SmartBASIC Data Pack. The following display should appear:



```
                 Coleco SmartBASIC V1.0






 ]_
```

Open the Digital Data drive and remove the Data Pack. Use the RESET COMPUTER button to return to the typewriter mode. This concludes our set-up procedure.

INSTALLING EXPANSION MODULE #3 WITH THE COLECOVISION SYSTEM

Unpack the expansion module components from their shipping carton. Check the parts against the list below and make sure they are all present.

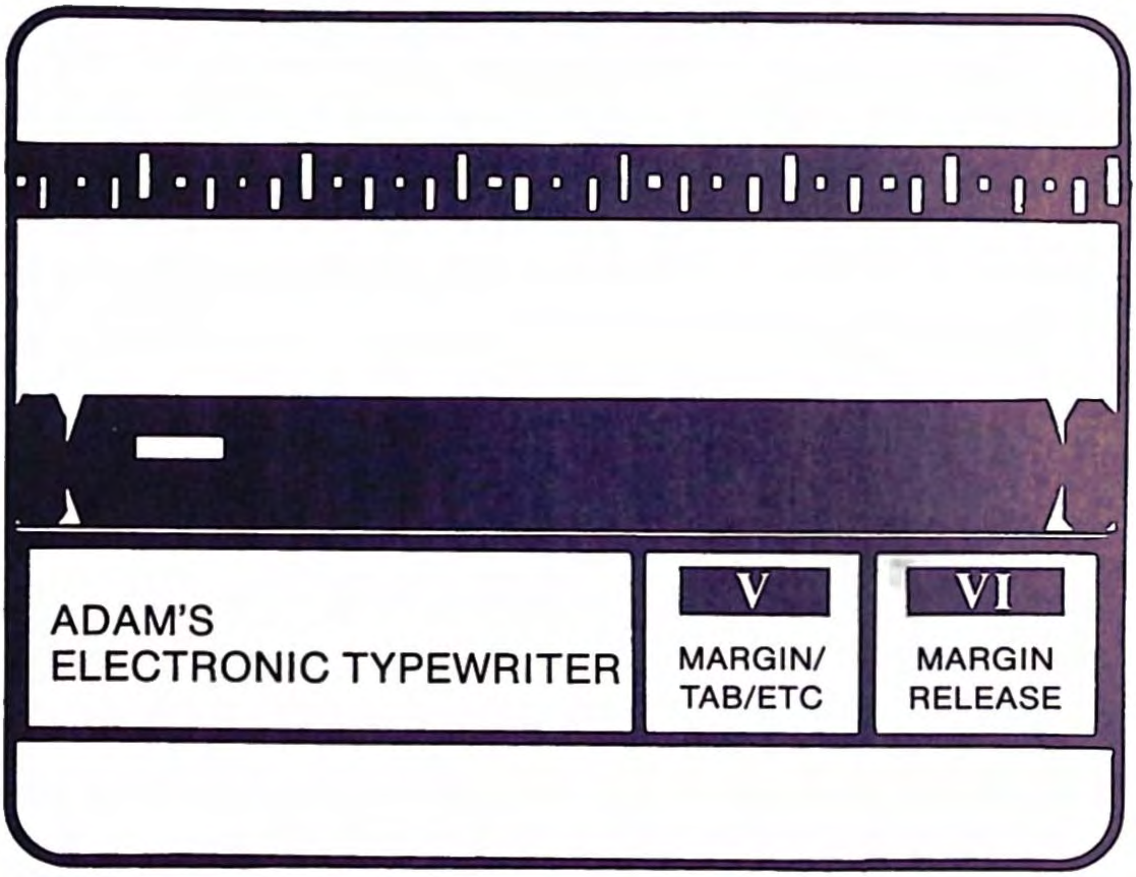
64K RAM Memory Add-on with one Digital Data Drive
System Interlock Tray
Keyboard Unit
1 Holder to attach game controller to keyboard
Daisy-Wheel Printer:
    Pica 10 pitch daisy wheel (installed)
    Carbon Ribbon Cartridge (installed)
3 Digital Data Packs:
    SmartBASIC Digital Data Pack
    Blank Digital Data Pack
    BUCK ROGERS Planet of Zoom Super Game Pack
Cords:
    Keyboard to Memory Add-on Unit
    TV switch box to system unit
    Printer to system unit
Manual:
    ADAM Set Up Manual
    ADAM Word Processing Manual
    ADAM SmartWriter Easy Reference Guide
    ADAM SmartBASIC Manual
    ADAM Super Game Pack Manual
3-to-2 Prong Plug Adaptor

DIRECTIONS FOR EXPANDING COLECOVISION TO INCLUDE ADAM

Before beginning, turn the system and TV off and make sure all cartridges have been removed.

1. Disconnect the power supply from the ColecoVision unit and store it. The expansion unit includes a new power supply.
2. Fasten the ColecoVision game unit to the system interlock tray.
3. Open the expansion module door on the front of the ColecoVision system and lock it in the open position.
4. Slide the expansion module into the ColecoVision system and lock the two units together.

5. Connect the keyboard to the system unit with the cord provided.
6. Connect the Controller Unit to the keyboard and attach one of the joysticks.
7. Connect the printer to the system console with the cord provided.
8. Switch the antenna box to the "game" position.
9. Plug the printer into the nearest 110/120 volt wall outlet, using the 3-to-2 prong adaptor if necessary. Make sure the outlet is grounded.
10. Turn your television to channel 3 or 4 (for best results, use the channel with the weaker reception). Make sure the selector on the back of the expansion unit agrees with the TV.
11. Turn the daisy wheel printer on with the switch located on the back. The following display should appear on the screen:

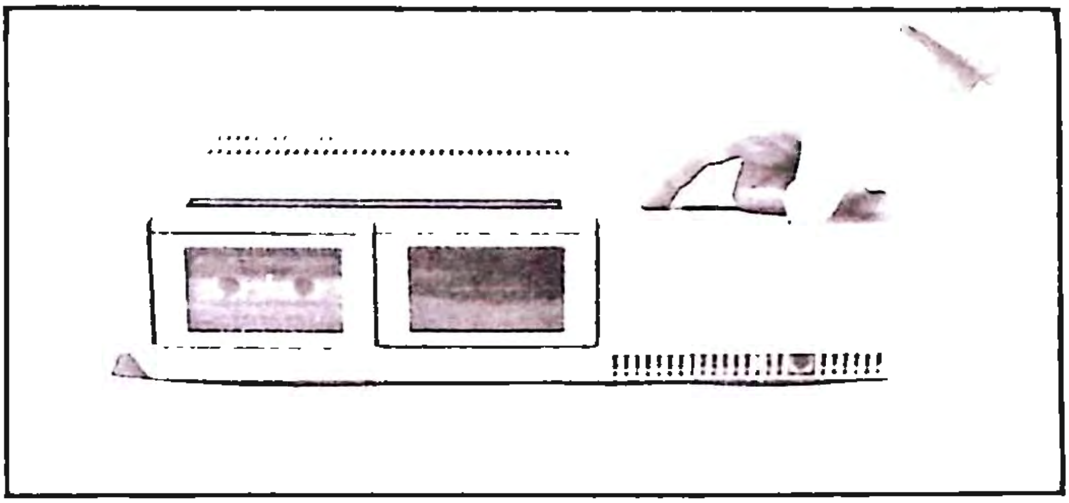


12. Insert a sheet of paper in the daisy wheel printer in the same manner as with a typewriter. If the paper is not straight, loosen the paper release and re-align it.

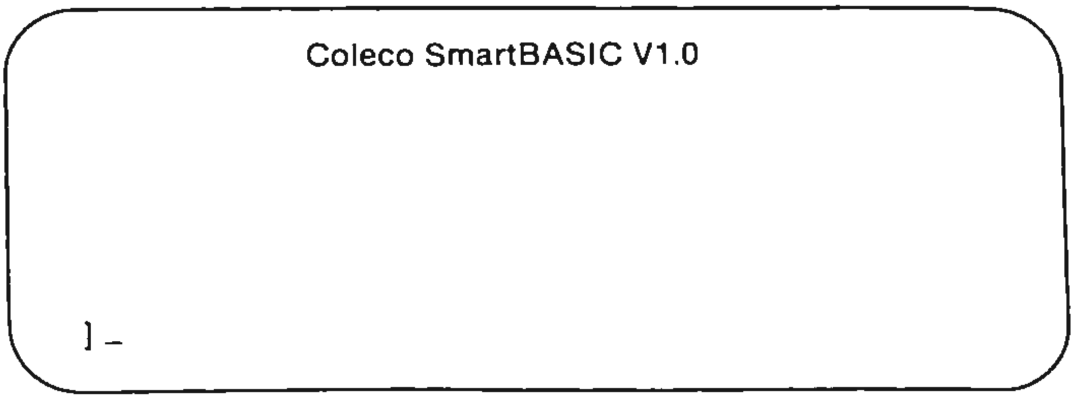
13. Type the following:

    Pack my box with twelve dozen liquor jugs

    The text should appear both on the screen and the printer.
14. Clear the screen by pulling the RESET COMPUTER switch forward, as shown below:



15. Insert the SmartBASIC Digital Data Pack into the Digital Data Drive and pull the RESET COMPUTER switch again. The Digital Data Drive should activate and load the program. This procedure takes approximately 30 seconds. **Do not interrupt the loading procedure.** Doing so might damage the SmartBASIC Data Pack. The following display should appear:



```
                    Coleco SmartBASIC V1.0




 ]_
```

Open the Digital Data drive and remove the Data Pack. Use the RESET COMPUTER button to go back to the typewriter mode. This concludes our set-up procedure.

If Something Goes Wrong . . .

ADAM has been designed to offer many of hours of trouble-free service, but occasionally hardware components fail. The remainder of this chapter is devoted to determining problems which may occur, and offering advice on how to remedy them.

NO POWER

Probably the most frightening type of failure is when the machine is turned on, and absolutely nothing happens. If this should occur, first check to see that the power cord is properly connected. If the unit still does not work, check the wall socket with an appliance that you know is functioning. If the wall outlet checks out, check your television set to make sure that it is functioning correctly and tuned to the proper channel. If your television is in good working order, then contact the ADAM HELP Hotline. The toll-free number is 1-800-842-1225 and may be contacted from 8:00 am to 5:00 pm Eastern Standard Time.

PRINTER PROBLEMS

The SmartWriter letter-quality printer has a built-in microprocessor which minimizes the number of moving parts in its design. This was a wise move on the part of ADAM's makers, since moving parts are generally more subject to failure from wear and misalignment. In this section, the various moving parts that can be adjusted without special tools will be reviewed. This discussion will include some commonly encountered problems and possible remedies, As always, caution should be exercised when working with electrical or moving parts. It is always a good precaution to unplug the unit while servicing it.

While the cable which connects ADAM to the printer is generally not regarded as a moving part, it is subject to movement. If cable movement causes the special plug to move, it may not be able to "communicate" effectively. Any time that the printer begins to type nonsense, the cable should be checked to see that it is securely seated.

The daisy wheel, shown in Figure 2-1, is made of a lightweight rigid plastic. Each of its spokes contains a single character whose imprint is transferred to paper through a inked ribbon by impact from a small metal hammer located at the top of the daisy wheel print mechanism. The wheel is aligned by means of a pin, as shown in Figure 2-2. If the wheel is not properly aligned, then the hammer will probably strike somewhere other than dead center. It is not unlikely that spokes would be broken with the wheel in this condition. The wheel must also be firmly seated. If not seated properly, it will wobble like a poorly thrown frisby, and cause additional wear on the drive motor and possible damage to the wheel itself.

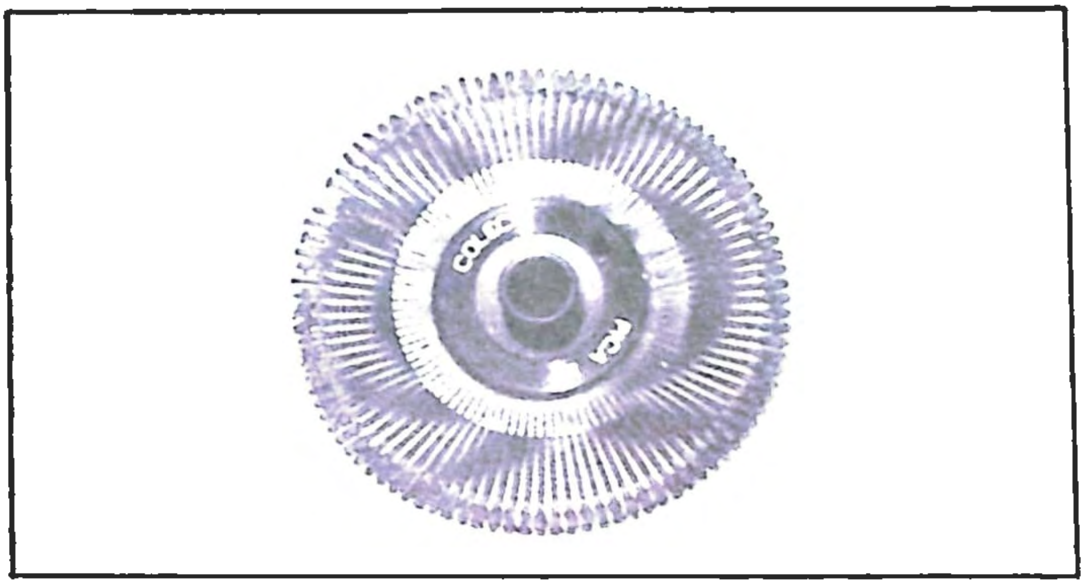


FIGURE 2-1. Pica Daisy Wheel

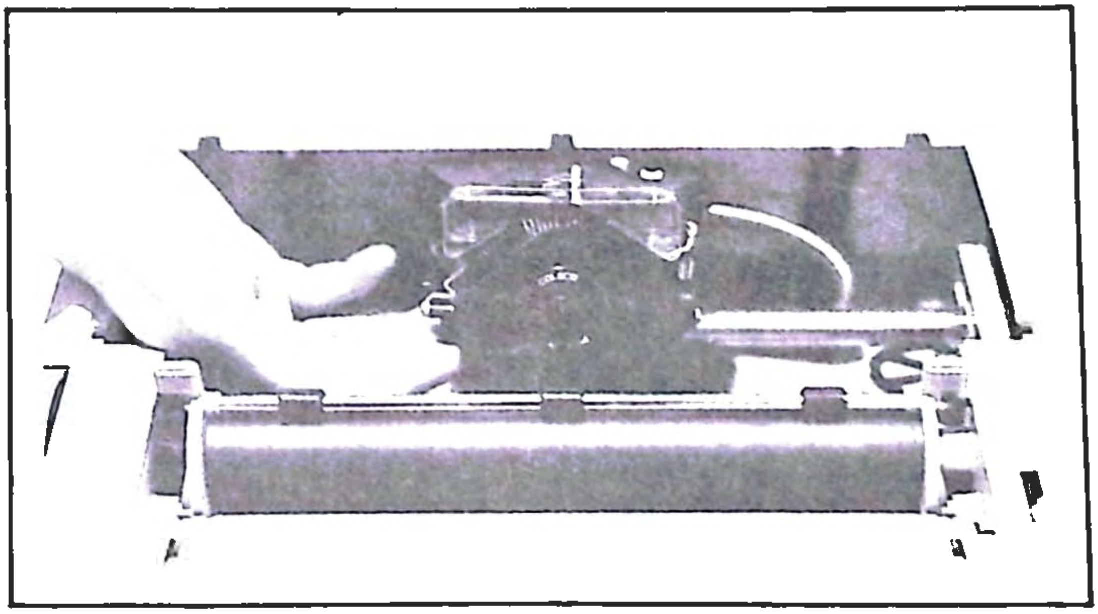


FIGURE 2-2. Daisy Wheel Alignment

The daisy wheel print mechanism both rotates the wheel to the proper position and impacts it for character transferral. It may be rotated forward by releasing the latches located on either side, as shown in Figure 2-3. The mechanism is re-secured by simply rotating forward and pressing down until the latches click back into place. If the mechanism is not firmly in place, the hammer will not have a solid surface to strike against. Additionally, the hammer will travel forward farther than normal and possibly break a daisy wheel spoke.

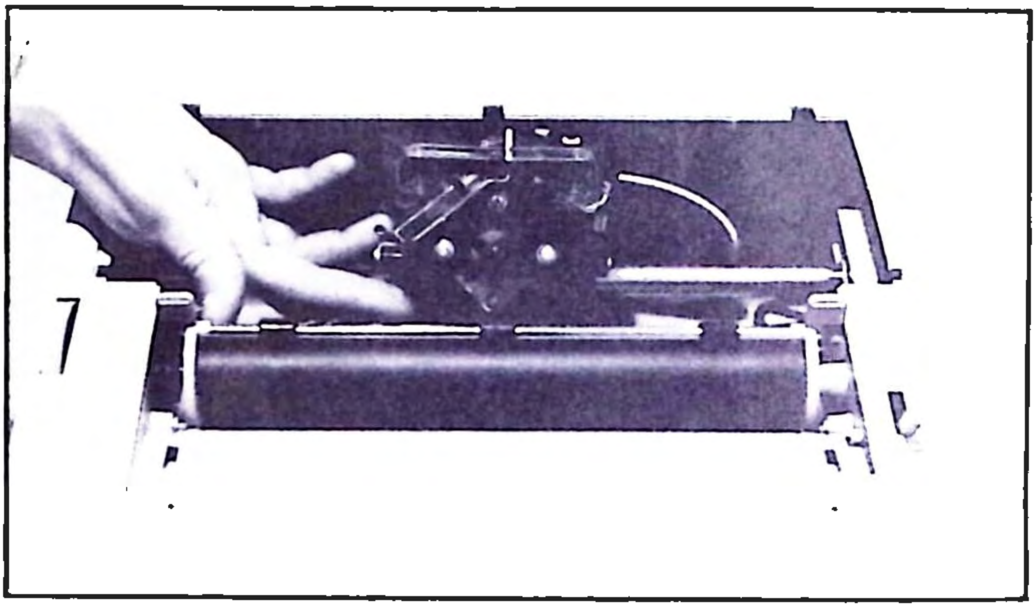
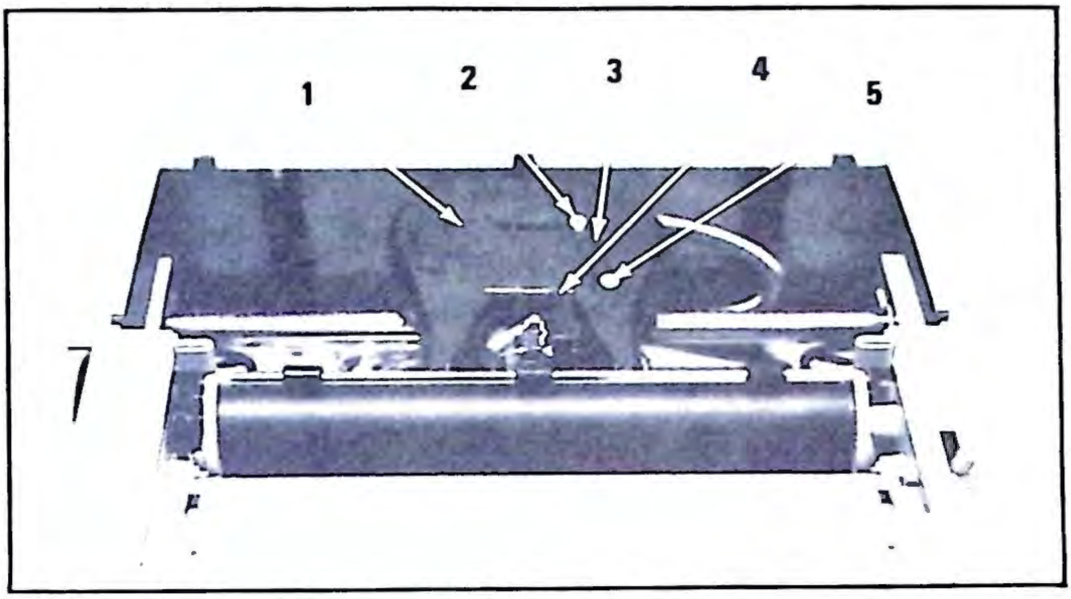


FIGURE 2-3. Releasing the Daisy Wheel Mechanism

The ribbon cartridge constantly gives the printer a fresh supply of inked ribbon. The ribbon is supplied from right to left by a spool on the right hand side of the cartridge (looking down from the front of the printer) which unrolls counter-clockwise. A pair of rollers pull used ribbon into the left hand side of the cartridge where a second spool turns clockwise to collect it. The rollers are driven from beneath by the print mechanism. A small rubber pulley transfers the motion of the rollers to the collecting spool. The ribbon cartridge is fastened to the print mechanism with a notched latch, shown in Figure 2-4.



1. Supply Spool 2. Collection Spool Collection Pulley 4. Notched Hold Down Latch 5. Ribbon Advance Roller

FIGURE 2-4. Ribbon Cartridge Assembly

If the printer is impacting, but no characters or poorly defined characters are being formed, then one of the aforementioned parts is probably the cause. Remove the printer cover to inspect the ribbon alignment. Make sure that the ribbon feeds around the right spring, between the daisy wheel and roller, and around the left spring.

If the alignment is correct, check to see that the cartridge has not run out of ribbon. Most cartridges have a slot on the feed side which allows the user to see how much ribbon remains. If the cartridge is not empty, type some text and watch to see if the ribbon advance roller rotates.

If the advance roller does not rotate, the phillips-head drive from below may not be engaged. Rotate the drive clockwise with the wheel located directly beneath it until a click can be heard. If the drive is engaged but will not advance the ribbon, then the ribbon is probably jammed.

Remove the cartridge and try advancing the ribbon either by rotating the advance roller from above or from below with a phillips-head screwdriver. If the ribbon does not advance, replace it. The cartridge will jam if the rubber pulley is not installed, so be sure it is in place when a new ribbon is being installed.

If the advance drive rotates manually and the ribbon also advances, try entering some text again. If the wheel does not advance on its own, adjust the notched latch so that it exerts more downward pressure on the ribbon cartridge. If this still does not remedy the problem, call the ADAM HELP hotline.

Sometimes in the process of printing, the print mechanism carriage may jam, causing the printer to begin printing lines in the wrong column. If this should happen, save whatever data is being processed, remove all Digital Data Packs, and pull the RESET COMPUTER switch. Check to see that the problem has been fixed by entering text in the typewriter mode. If this does not work, turn ADAM off with the switch behind the printer and gently push the print mechanism to the left hand side. If the printer does not return to normal when the unit is turned back on, call the ADAM HELP hotline.

SMARTBASIC OR APPLICATIONS PROGRAM WILL NOT BOOT

Generally, when a SmartBASIC Data Pack is present in a Data Drive, the COMPUTER RESET switch causes SmartBASIC to be "booted" or loaded into memory. If this procedure does not produce the desired results, make sure that the proper Data Pack is in use. Also, be sure that the Data Drive door has been closed properly. If the problem persists, the Data Pack or the drive mechanism is probably at fault. The BUCK ROGERS™ game can be used to test the Digital Data Drive. If the game runs as it should, the SmartBASIC Data Pack is probably damaged. For assistance, call the ADAM HELP hotline.

One thing to remember when handling Digital Data Packs is that a computer program will not run if it contains any errors. Scratches are annoying on stereo records, but deadly to data cassettes. We mentioned the need for careful handling of Data Packs in Chapter 1, and feel that it bears repeating here:

> **Never touch the surface of a Digital Data Pack's tape. Keep it away from all magnetic fields. *Never* turn ADAM off when a Digital Data Pack is still in it's drive.**

3

The SmartWriter Word Processor

Introduction

You have undoubtedly heard by now how word processors are changing the way people work and think, but maybe you have wondered exactly what a word processor is. A word processor is a computer program which allows a user to create, manipulate, store, and retrieve text. A word processor can be used to write a report, a memo, or even a book.

SmartWriter is a powerful word processor which allows the user to create, edit, and print letter quality text on the ADAM home computer. SmartWriter is contained on a ROM (Read Only Memory) chip and can be made available with a single keystroke.

A powerful menu-driven help system assists the user at every stage of writing. By simply looking at the bottom of the screen, the user can examine available options. Each time, a fresh menu of useful alternatives will appear.

With SmartWriter, the user can sit down at the keyboard and enter text just as he or she would at a typewriter. Unlike a typewriter, however, SmartWriter allows text editing. Later, after phrases have been entered, the user can correct spelling or grammar or change wording as desired.

Text can be formatted in an almost limitless number of ways. Because most people use 8½ x 11 inch paper with 1 inch margins on the top and bottom, SmartWriter **defaults** its formatting options to reflect this. Through the guidance of the menu system, these defaults can be changed.

SmartWriter enables ADAM to store text as a **file** on a Digital Data Pack. You will soon learn to browse through the Digital Data Pack's file directory, to bring files into the SmartWriter environment, and to print the contents of any file.

The remainder of this chapter will explain, by way of examples, SmartWriter usage. The best way to become familiar with SmartWriter is to scan this chapter. Then, sit down and **work through the examples.**

GETTING STARTED

Start ADAM with the power switch located on the back of the printer. Since SmartWriter is contained on a ROM chip, it does not have to be called up from a Digital Data Pack.

Start SmartWriter by pressing the ESCAPE key located in the upper left hand corner of the keyboard. The first thing SmartWriter does is to present the **Standard Format Screen** as shown in Figure 3-1.

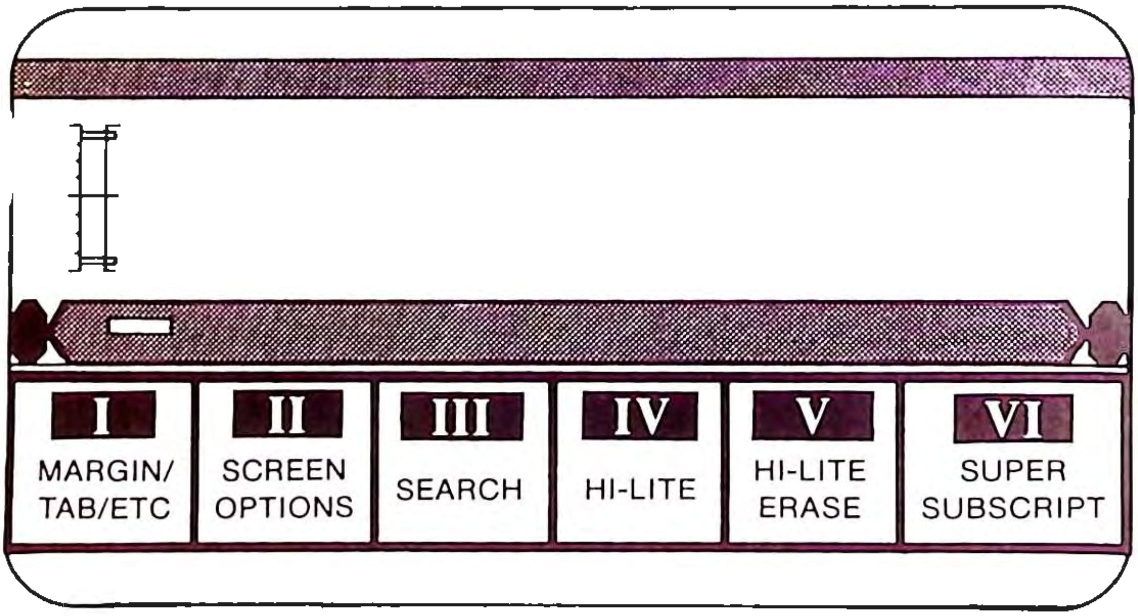


FIGURE 3-1. The Standard Format Screen

At the bottom of the screen, a figure resembling a typewriter roller, or **platen**, will be displayed. An underline, known as a **cursor,** will appear to the left of the platen. The cursor's initial position upon SmartWriter startup is known as the **home position.**

The top of the screen will contain a **horizontal margin scale.** The small white dots that appear every 5 spaces are the tabs. The red marks at positions 10 and 70 are the **horizontal margin markers.**

The leftside of the screen will contain a **vertical margin scale.** At its very top should be the number 11, indicating standard 11-inch paper. The marks visible at the top and bottom of the vertical margin scale indicate the vertical margins and are referred to as **vertical margin markers.** As with the horizontal scale, the large white mark indicates the **vertical cursor position.**

Figure 3-2 shows the Standard Format Smart Key Labels. We will discuss the use of the smart keys shortly.

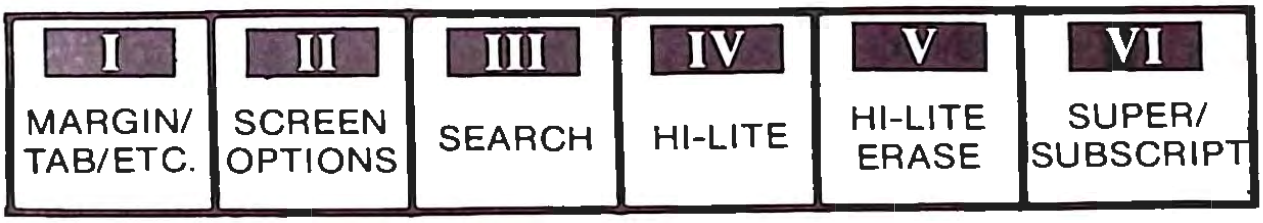


FIGURE 3-2. Standard Form Smart Key Labels

THE KEYBOARD

Figure 3-3 illustrates the ADAM keyboard. In addition to the standard typewriter keys, extra keys are located above and to the right of the main keyboard.

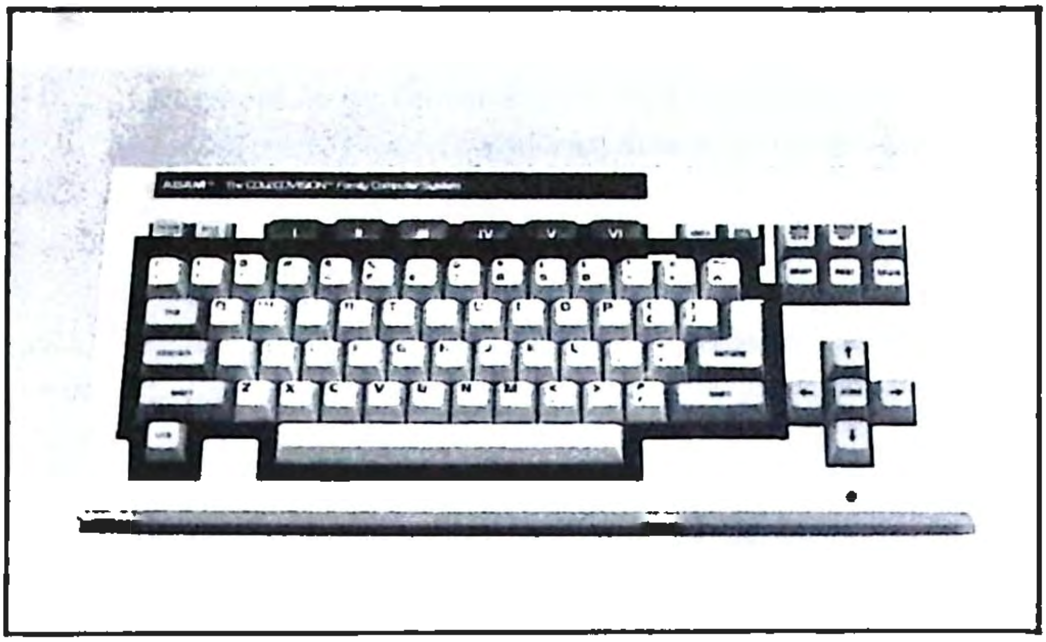


FIGURE 3-3. The ADAM Keyboard: Standard Typewriter Keys

The keys in the top row are known as command keys (Figure 3-4). These keys instruct ADAM to perform a variety of tasks. One of these, the ESCAPE/ WP key, has already been used to load SmartWriter. The other keys will be introduced shortly.

Notice that the keys labelled I through VI on the top of the keyboard are reproduced on the display, and that a command is shown beneath each numeral. These keys are known as **smart keys.** The smart keys are unique because their functions change as SmartWriter performs different tasks. Changing functions in this way allows a number of commands to be entered in a single keystroke. The smart key functions change in concert with SmartWriter so as to be of maximum benefit to the user.

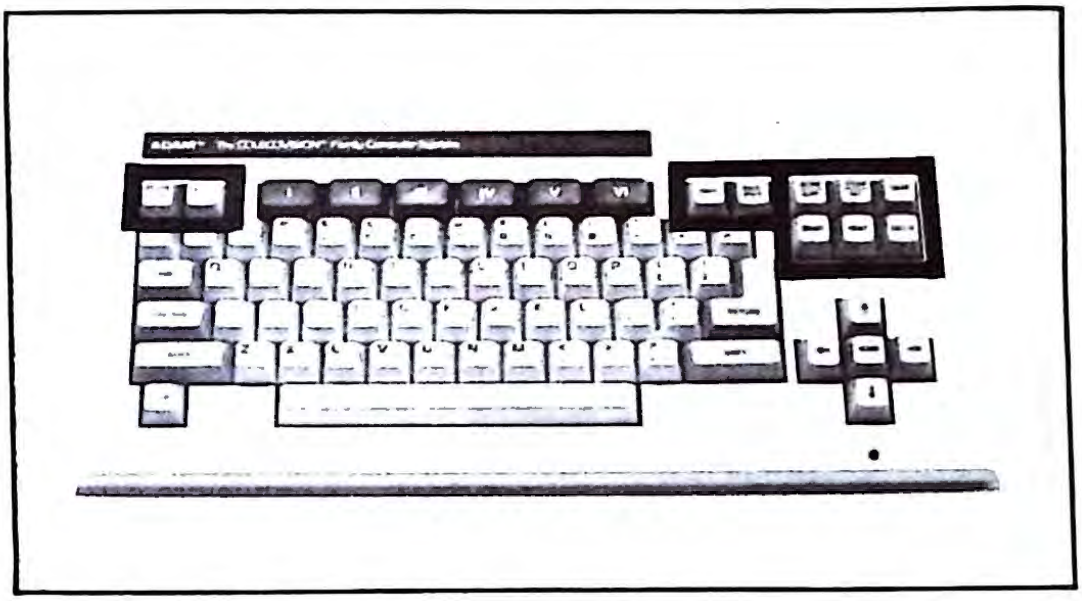


FIGURE 3-4. The ADAM Keyboard: Command Keys

The last group of keys, located in the lower right hand corner of the keyboard, are known as **cursor keys** (Figure 3-5). These keys enable cursor movement in the direction indicated by the arrows, one character at a time, without changing the existing screen text. The **HOME** key, located in the center, returns the cursor to the home position.

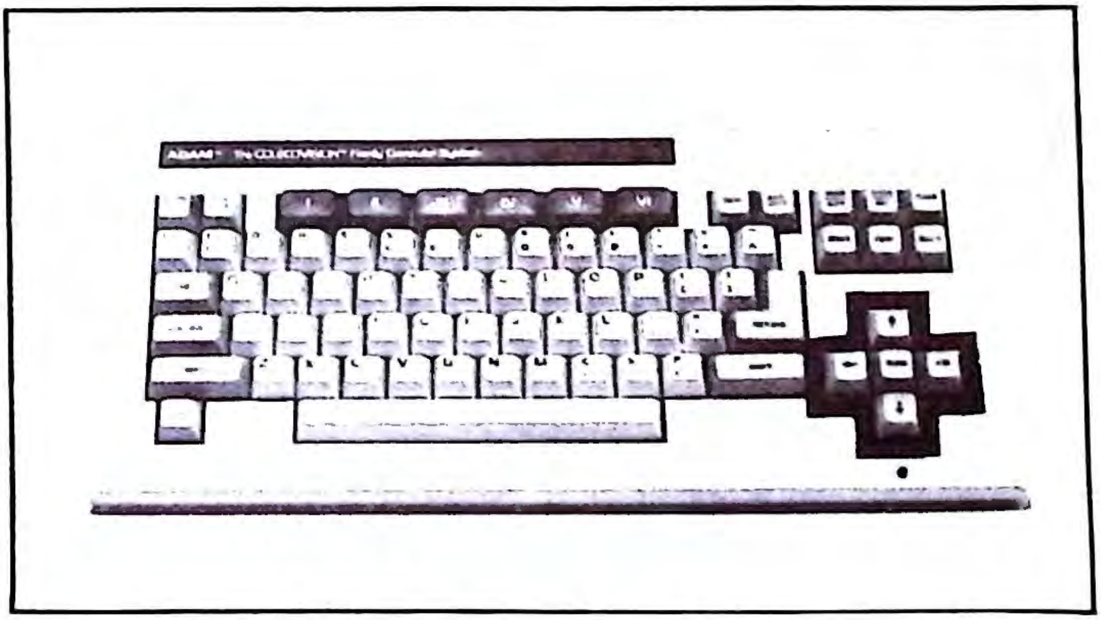


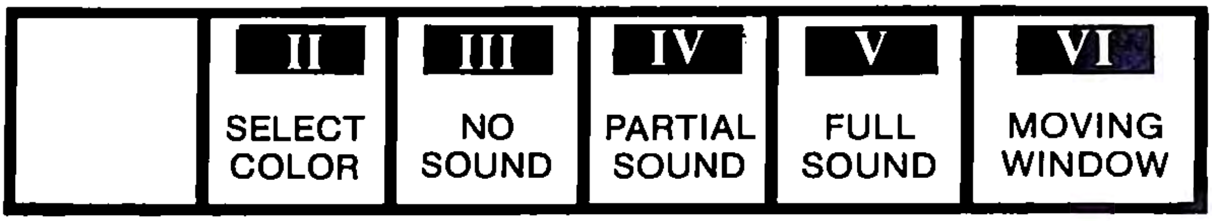
FIGURE 3-5. The ADAM Keyboard: Cursor Keys

Screen Options

In addition to the standard format screen, there are a number of alternative screen formats. SmartWriter allows the user to select from a number of color, sound level, and screen layout options.

SOUND

SmartWriter allows the user to select the desired audio level quickly via key II: SCREEN OPTIONS. The following will be displayed when this key has been pressed.

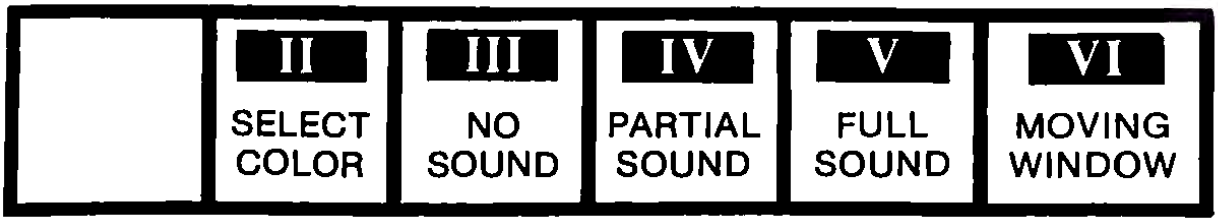


	II	III	IV	V	VI
	SELECT COLOR	NO SOUND	PARTIAL SOUND	FULL SOUND	MOVING WINDOW

Press key V to obtain full sound. With this option in force, an audible sound will be emitted with every keystroke. Press key III (NO SOUND) to suppress this feature.

COLOR

When SmartWriter is turned on, a blue screen will appear. To obtain a different background color, first press key II (SCREEN OPTIONS). As before, the keys change to:



	II	III	IV	V	VI
	SELECT COLOR	NO SOUND	PARTIAL SOUND	FULL SOUND	MOVING WINDOW

Press key II (COLOR SELECT) — the smart labels will change to:

I	II	III	IV	V	VI
WHITE	GREEN	BLACK	GREY	BLUE	DONE

If a color monitor is in use, pressing keys I through V will display the color schemes ADAM has to offer. You may, for example, prefer WHITE (key I) for normal text entry or BLACK (key III) for slide preparation. Whatever choice is made, it can be placed into effect by pressing key VI (DONE).

THE MOVING WINDOW FORMAT

SmartWriter offers an alternative to the screen format used thus far. In the **moving window format,** the screen acts like a window through which 20 rows and 36 columns of text will be visible. The main advantage to this format is that text will be displayed exactly as it will be printed. Figure 3-6 demonstrates the moving window.

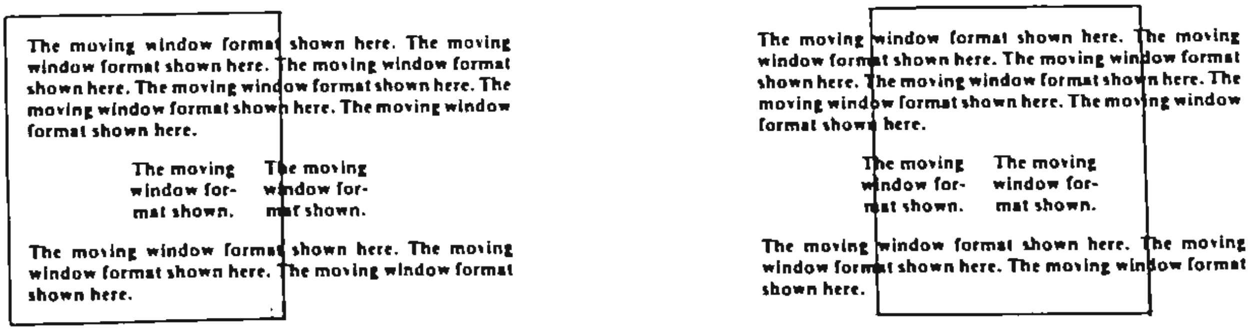
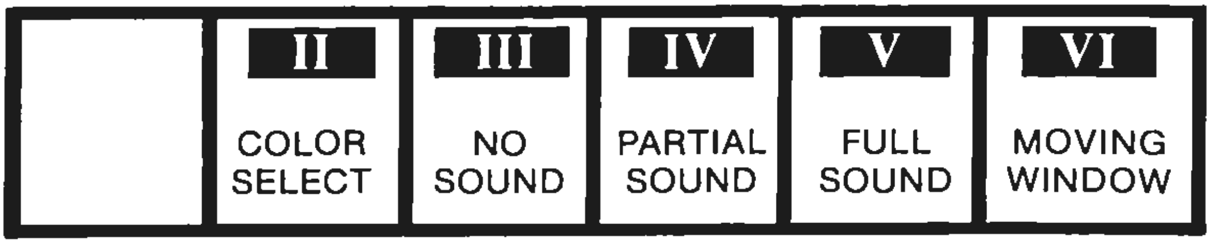

The moving window format shown here. The moving window format shown here. The moving window format shown here. The moving window format shown here. The moving window format shown here. The moving window format shown here.
The moving window for- mat shown.
The moving window for- mat shown.
The moving window format shown here. The moving window format shown here. The moving window format shown here.
The moving window format shown here. The moving window format shown here. The moving window format shown here. The moving window format shown here. The moving window format shown here. The moving window format shown here.
The moving window for- mat shown.
The moving window for- mat shown.
The moving window format shown here. The moving window format shown here. The moving window format shown here.

FIGURE 3-6. The Moving Window Format

To obtain the moving window format, press key II (SCREEN OPTIONS). The smart keys will change to:



	II	III	IV	V	VI
	COLOR SELECT	NO SOUND	PARTIAL SOUND	FULL SOUND	MOVING WINDOW

Press key VI (MOVING WINDOW). The platen will disappear, and the text will move to the top of the screen. The cursor should be positioned under the first letter in the first row. Move the cursor to the right with the → cursor key and notice that when the edge of the screen has been reached, the text will "jump" to the left four characters, leaving the cursor position within the text unchanged.

Format changes are largely a matter of personal preference. Many users find, for example, that table creation is easier in the moving window format while jotting a short note is easier in the standard format.

To return to the standard format, press SCREEN OPTIONS (II) and then STANDARD FORMAT (VI).

File Handling

In pre-computer days, letters or memos were created with a typewriter, pen, or pencil and a piece of paper. Once a large collection of letters was amassed, some form of organization, storage, and retrieval became necessary. The most conventional form of organization was the file cabinet. With a computer, data can be entered at a keyboard and stored on some form of magnetic media. A form of computer storage has evolved which mimics a file-cabinet type of organization.

CREATING AND STORING A FILE

Enter the following text on the keyboard:

```
Whether 'tis nobler in the mind to
suffer the slings and
arrows of outrageous fortune, or to
take arms against a sea
of troubles, and by opposing end
them?
```

Type quickly and if a mistake is made, ignore it for now. We will return later and fix it.

Notice that pressing the RETURN key was not necessary when the end of a display line was encountered. This feature in known as **word wrap.** With it, entire paragraphs can be entered without ever having to hit the RETURN key.

Notice also that each line is broken into two parts. Since your screen only allows 40 characters, this feature allows the user to view an entire standard line on your screen.

To save data written as a file, a Digital Data Pack must be inserted into its drive, as shown in Figure 3-7.

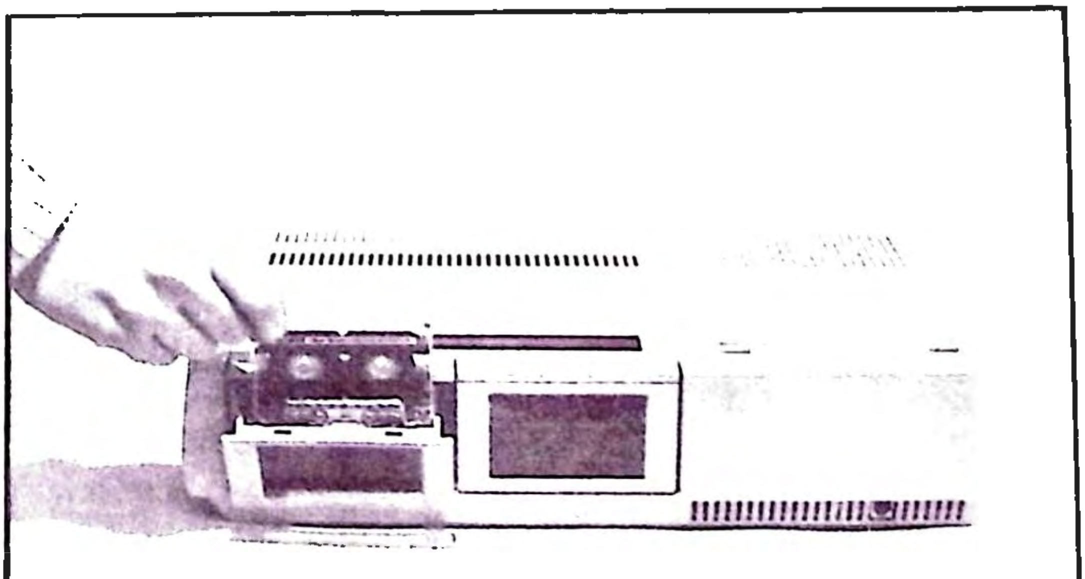
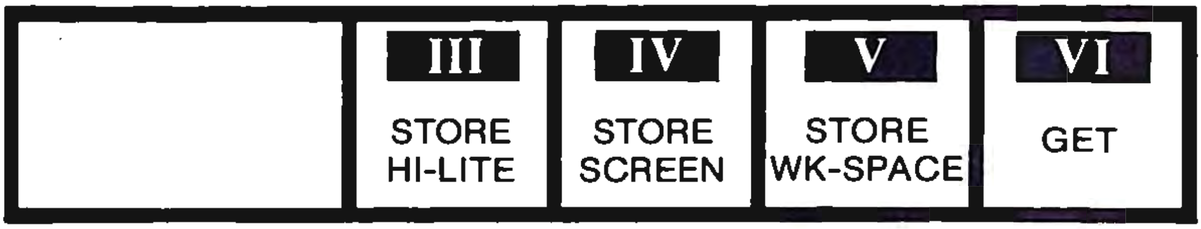
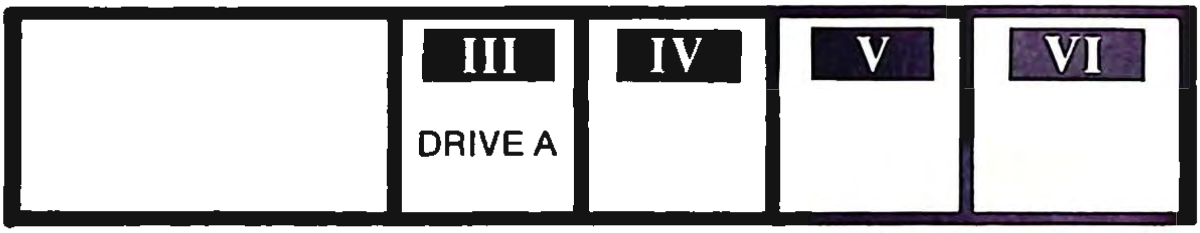


FIGURE 3-7. Inserting the Digital Data Pack

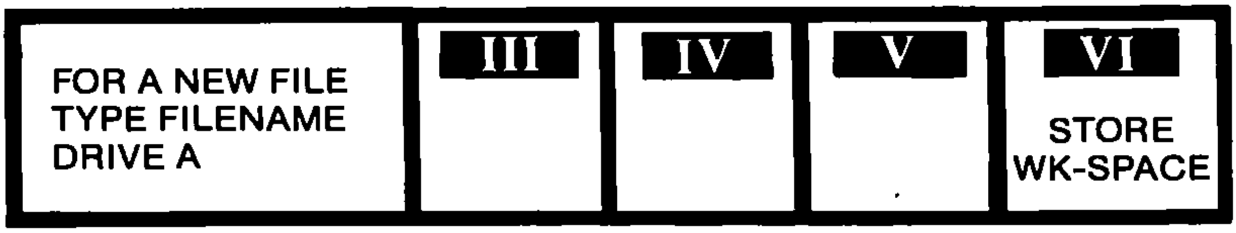
Next, press the STORE/GET command key located in the upper right-hand corner of the keyboard. Notice that the labels below the smart keys changed as shown below. The old commands are no longer applicable. The new commands denote the current options.



Press key V (STORE WK—SPACE) and note that the smart keys change again as shown below.



This new set of keys allow data to be stored in a number of ways. Press key III (DRIVE A) to prepare ADAM to write data on the Digital Data Pack. The smart key display will change again. This time a filename will be requested. Data will be stored under the name chosen. Any name may be used as long as it contains ten characters or less. It is often helpful to use a **mnemonic** name, that is, a name that serves as a reminder of the file contents. Enter HAMLET for this example.

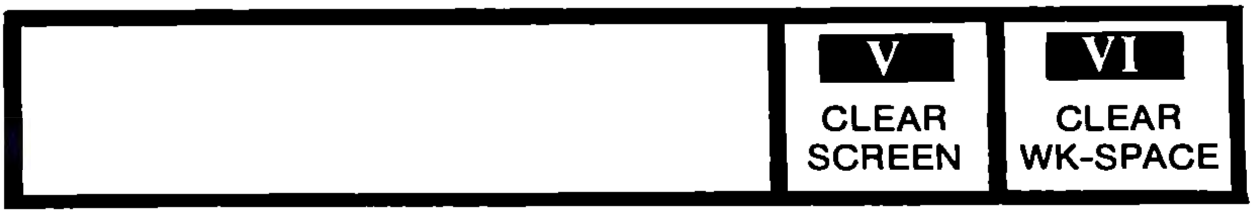

FOR A NEW FILE
TYPE FILENAME
DRIVE A

To activate the storage procedure, press key VI (STORE WK-SPACE). A soft whirring sound will be audible as the data is recorded on the Digital Data Pack.

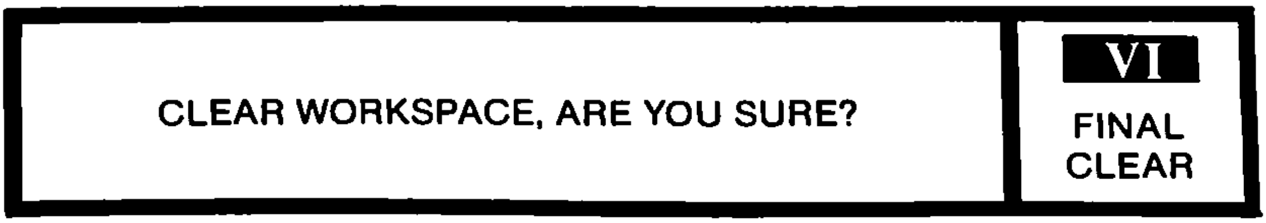
RETRIEVING A FILE

Now that HAMLET has been written to the Data Pack as a file, ADAM can be powered off without the permanent loss of the data in HAMLET. Remember to remove the Data Pack before turning the computer off.

To demonstrate that HAMLET has been stored, first ADAM's memory will be cleaned. Then, HAMLET will be reloaded into ADAM's memory from the Digital Data Pack. To clear ADAM's memory, press the CLEAR command key. The smart keys will change as shown below.

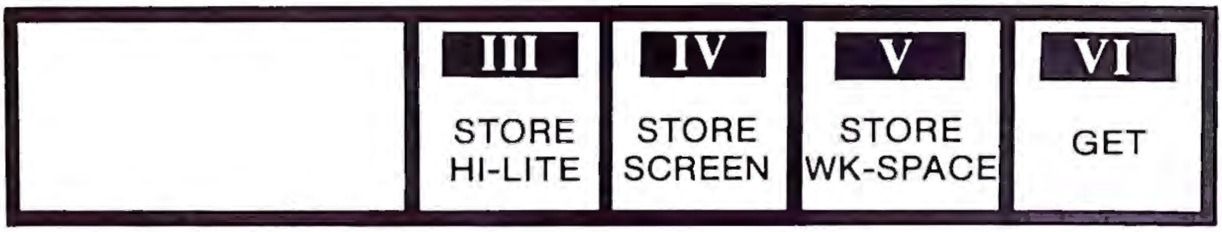


Press key VI (CLEAR WK-SPACE). SmartWriter will display a prompt asking the user to verify that the workspace contents are to be erased. Reply by pressing key VI (FINAL CLEAR). The screen will return to the standard format.

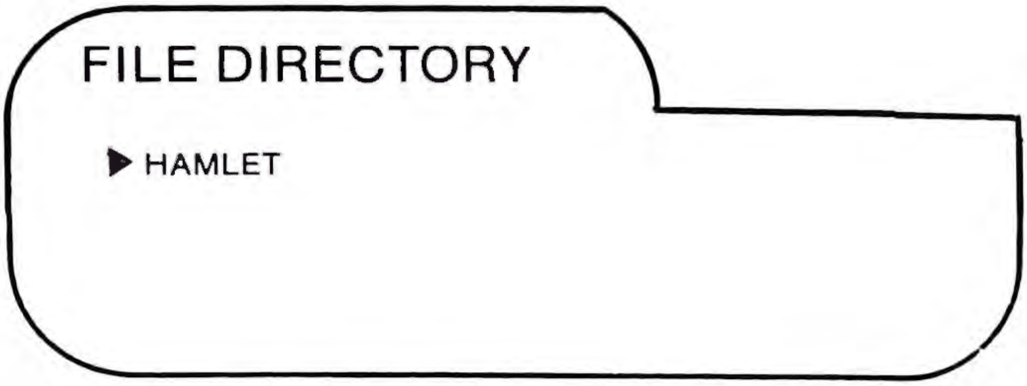

CLEAR WORKSPACE, ARE YOU SURE?

Insert the Digital Data Pack into its drive. Execute SmartWriter by pressing ESCAPE/WP.

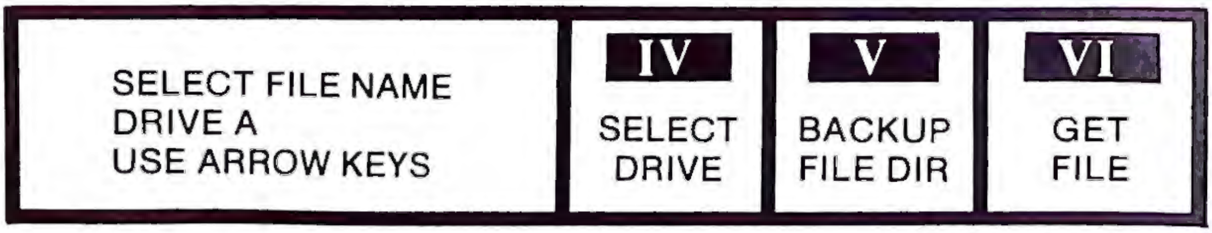
To recall the file, press the STORE/GET key. As always, the smart key display changes to reflect available options:



Press key VI (GET), followed by key III (Drive A). The Digital Data Drive will whirr for a few seconds and a **file directory** of drive A will be presented. The directory should only contain the entry HAMLET as shown below:


FILE DIRECTORY

▶ HAMLET


SELECT FILE NAME
DRIVE A
USE ARROW KEYS

To recall the file, first use the cursor movement keys to move the cursor to the file entry HAMLET. Read the file into memory by pressing key VI (GET FILE). The text entered earlier should re-appear.

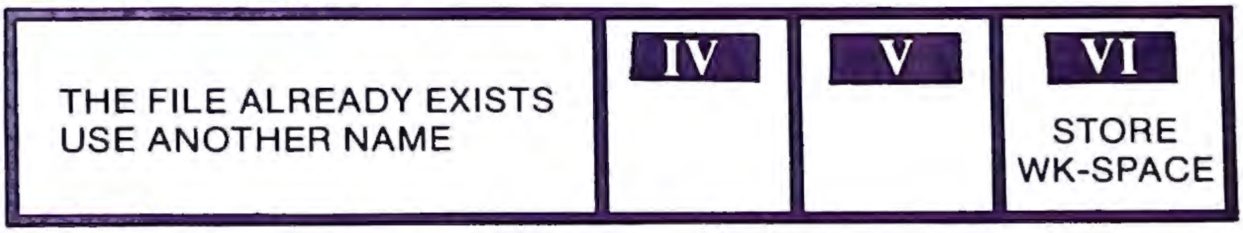
BACKUP FILES

It is a good practice to create a backup of a file before editing the original. The SmartWriter designers understood this and included a backup feature.

When HAMLET was first saved, only one version existed. A backup can be created by resaving the original file. This can be accomplished with the following sequence.

```
   STORE/GET
 V:STORE WK-SPACE
III:DRIVE A
VI:STORE WK-SPACE
```

Note that in the last step it was not necessary to enter the file name — SmartWriter uses the name of the **last file called** by STORE/GET. If you try to enter HAMLET, for example, SmartWriter will reply:

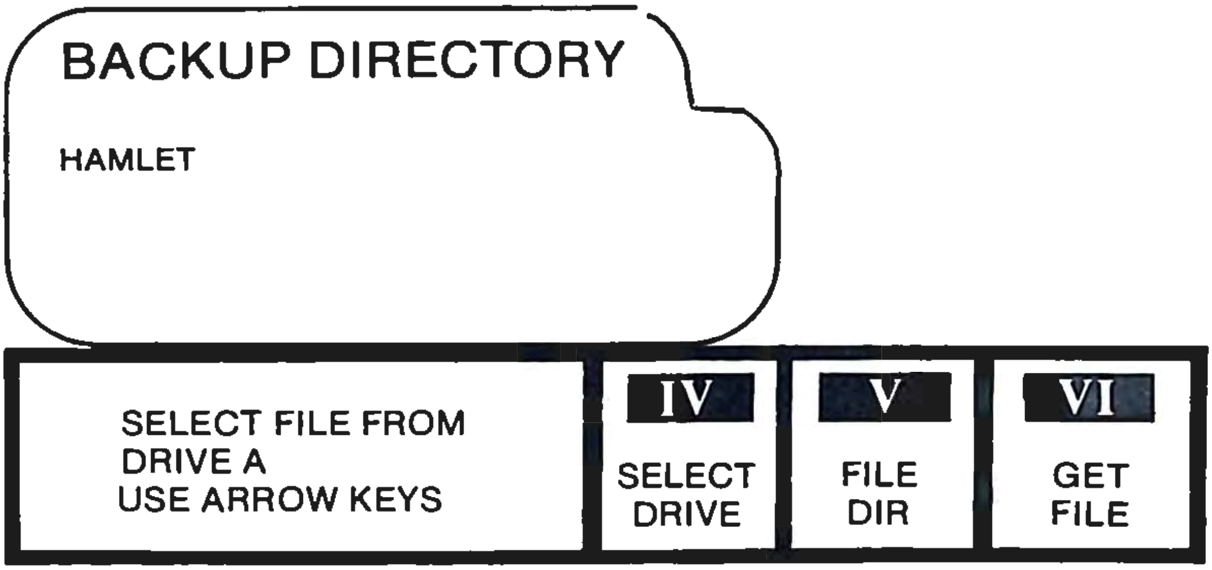


THE FILE ALREADY EXISTS
USE ANOTHER NAME

When you entered the backup sequence ADAM's Digital Data Drives should have engaged and written a file. To check this enter:

```
   STORE/GET
VI:GET WK-SPACE
III:DRIVE A
 V:BACKUP FILE DIR
```

You should see the following:



BACKUP DIRECTORY

HAMLET

SELECT FILE FROM
DRIVE A
USE ARROW KEYS

If another drive is connected, it can be selected by pressing key IV (SELECT DRIVE), to return to the main directory, press key V (FILE DIR). The backup file can be loaded into the workspace by pressing key VI (GET FILE).

Either the file or backup directory can be exited directly to Smart-Writer's workspace by pressing the ESCAPE key.

Since HAMLET is already present in our workspace and the backup is identical to the original, we can go back without having to read the file from the Digital Data Pack by pressing the ESCAPE key.

EDITING FILES

Once a backup has been created, a file can be safely edited. There are five general catagories of editing tools:

1) Cursor movement
2) Text replacement
3) Text deletion
4) Text insertion
5) Text movement

As will be evidenced shortly, these tools may be applied to single characters, words, blocks of text, or even files.

CURSOR MOVEMENT

The cursor control keys allow the cursor to be moved around a SmartWriter file. To use these keys, first bring the file named HAMLET into the workspace. Try experimenting with the cursor keys by themselves. The following will soon be evident:

↑ :Moves the cursor up by one line.
→ :Moves the cursor to the right by one character.
↓ :Moves the cursor down by one line.
← :Moves the cursor to the left by one character.
HOME :Moves the cursor to the top line of text on the screen.

The cursor movement keys offer a relatively efficient means of moving a file. However, SmartWriter allows more rapid movement by simultaneously pressing the HOME and arrow keys. For example, if the cursor is positioned at the beginning of a line and HOME + → is pressed (that is, press HOME and → **at the same time**), then the cursor should move to the right hand side of the screen. The fast movement combinations are shown below:

HOME + ↑ :Moves the cursor to the top of the platen and brings the top line of screen text onto the platen. In the moving window format, moves the cursor to the top of the screen.

HOME + → :Moves the cursor to the right hand side of the platen. In the moving window format, moves the cursor to the right edge of the screen.

HOME + ↓ :Moves the screenful of text below the platen onto the screen. In the moving window form at, the cursor is moved to the bottom of the screen.

HOME + ← :Moves the cursor to the left hand side of the platen. In the moving window format, it is moved to the left edge of the screen.

One important fact to bear in mind is that the cursor keys cannot actually create new lines, even though the screen may indicate that they have done so.

SmartWriter behaves differently in the moving window format than it does in the standard format, so cursor movement in both formats will be described. In the standard format, the cursor keys can be used to move beyond the last line of text entered. Additional text can then be added. That text will remain if the file is saved. However, any additional blank lines "created" with the cursor keys will vanish. To examine this, use the cursor keys to move seven lines down from "them?", and STORE this under the name SEVEN with:

```
   STORE/GET
 V:STORE WK-SPACE
III:DRIVE A
   SEVEN (You must type this)
VI:STORE WK-SPACE
```

Now, CLEAR the workspace with:

```
   CLEAR
VI:CLEAR WK-SPACE
VI:FINAL CLEAR
```

Bring SEVEN back into your workspace with:

```
   STORE/GET
VI:GET WK-SPACE
III:DRIVE A
   SEVEN (Use cursor keys)
VI:GET WK-SPACE
```

By moving to the end of the file with the cursor keys, it will be apparent that "seven" appears directly below "them?".

In the moving window format, the cursor keys can also be used to move beyond the last line of text entered. In this mode, neither the new lines nor the new text will be saved. CLEAR the workspace as shown above. Then, enter the following command sequence to display the directory:

```
   STORE/GET
VI:GET WK-SPACE
III:DRIVE A
```

Use the cursor keys to position the ► next to the file named SEVEN and delete it with:

```
   DELETE
VI:FINAL DELETE
```

Now, position the cursor next to HAMLET and bring it into the workspace with key VI (GET FILE). If the ADAM is not already in the moving window format, this format can be entered as follows:

```
      II:SCREEN OPTIONS
      VI:MOVING WINDOW
```

If text is added to HAMLET in the same manner described under the standard format, when the file is retrieved, it will be evident that nothing was added to it.

TEXT REPLACEMENT

To illustrate text replacement, use the cursor keys to move to the end of HAMLET. Add:

-to die, -to sleap

Wait — "sleep" is misspelled. To fix it, just bring the cursor to the "a" in "sleap" and press "e". The "a" will be replaced. Any text can be replaced in this manner by simply typing over the old text.

TEXT DELETION — BACKSPACE

To illustrate text deletion, add the following text to HAMLET:

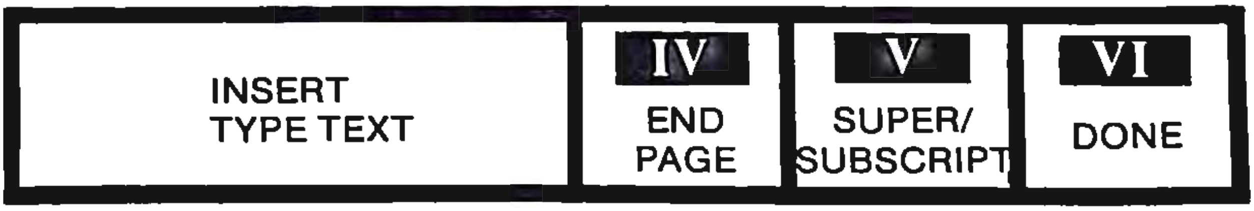
To sleep peacefully, perchance to dream-

The word "peacefully" doesn't belong. To delete it, position the cursor to the comma following "peacefully". Each time the BACKSPACE key is pressed, a character will disappear. Continue pressing BACKSPACE until "peacefully" has been deleted.

TEXT INSERTION

Suppose a line was to be added to HAMLET. The first step would be to position the cursor at the beginning of the file. This can be accomplished by pressing the HOME cursor key.

Next, press the INSERT command key to enter the insert mode. Existing text will be erased and the following smart key labels will be displayed.



INSERT
TYPE TEXT

Type:

```
To be or not to be, -that is the
question:-
```

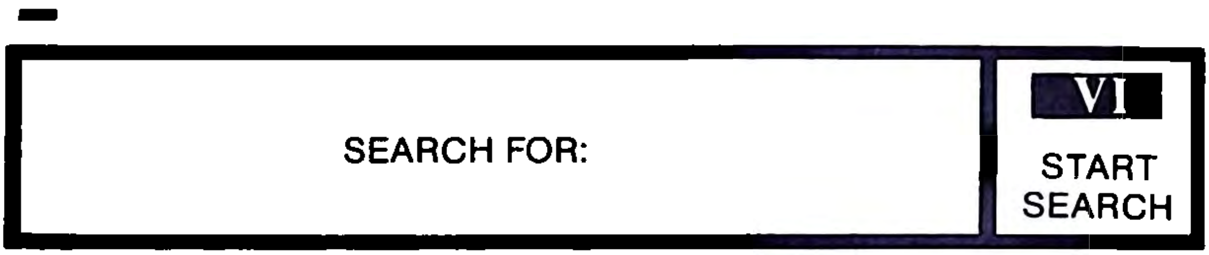
Then, enter the text by pressing key VI (DONE). The following text should then appear.

```
To be or not to be, -that is the
question:-Whether 'tis
nobler in the mind to suffer the
slings and arrows of
```

FINDING TEXT

Although the cursor keys can be used to move to a specified position within a file, generally key III (SEARCH) is a more efficient means of doing so. To illustrate the use of SEARCH, we will scan HAMLET for the word "Whether".

Go to the beginning of the file by pressing the HOME cursor key. Next, press key III (SEARCH). The smart keys will reappear as shown below:

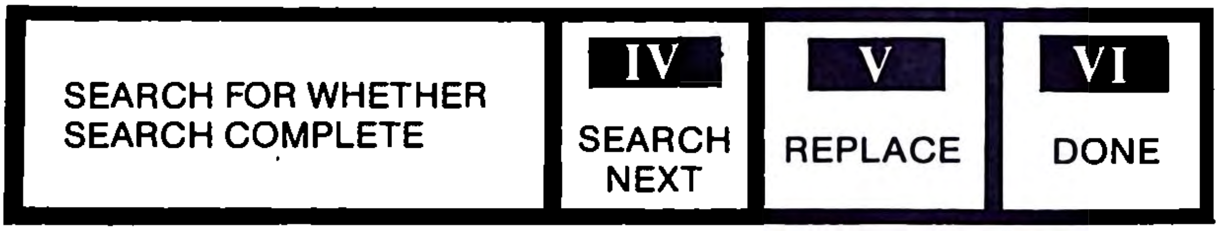


SEARCH FOR:

VI
START
SEARCH

Enter "Whether" and then press key VI (START SEARCH). The cursor should be positioned at the W in "Whether", as shown below:

```
To be or not to be, that is the
question:-Whether 'tis
```



SEARCH FOR WHETHER
SEARCH COMPLETE

IV
SEARCH
NEXT

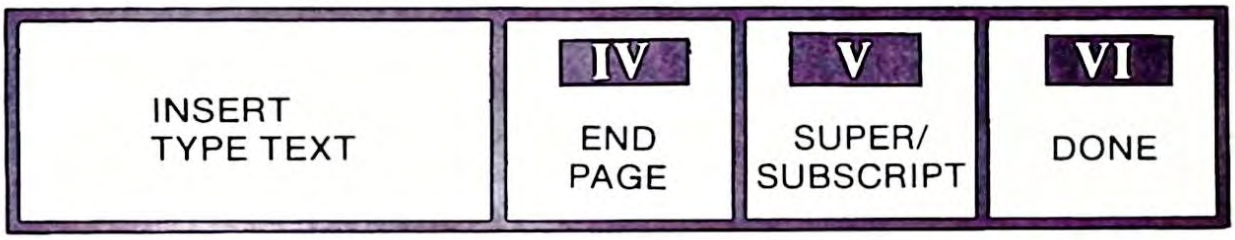
V
REPLACE

VI
DONE

In this case, we only want to find "Whether" once, so press key VI (DONE).

SEARCH does have some limitations. One of them is that it does not differentiate between upper and lowercase letters. For example, if MacDuff was the object of the search, the cursor would be positioned at MACDUFF, Macduff, or mACDUFF. The other limitation to SEARCH is that it will not accept non-alphabet characters. In other words, SEARCH could not be used to locate $10.00 or (#@&% !).

Getting back to our example, suppose that we wished to place "Whether" on a separate line. We could do so by inserting a carriage return. Our first step would be to press the INSERT key. The smart keys will change as follows:



Our next step is to press the RETURN key. This causes the symbol ◄ to appear just prior to "Whether". The insertion of the carriage return will cause the text beginning with "Whether" to appear on a new display line. Press key VI (DONE) at this point to exit the insert mode.

ESCAPE AND UNDO

Suppose that you intended to SEARCH (key III) for a word but accidentally pressed key II (SCREEN OPTIONS). Such errors occur regularly — a fact that SmartWriter designers recognized. To handle this type of error, simply press the ESCAPE/ WP key, and the standard menu will reappear.

To illustrate the usage of UNDO, move through HAMLET until the cursor has been positioned over the "W" in "Whether". This can be accomplished by using either the cursor control keys or the SEARCH feature. Use the BACKSPACE key to delete several characters. If the UNDO key is subsequenly pressed, "Whether" will reappear.

Note that UNDO only works on text changes. If, for example, the margins were reset and UNDO was pressed, nothing would happen. This type of error must be repaired by re-entering the correct MARGIN/-TAB/ETC sequence.

TEXT MOVEMENT

It is often useful to move a block of text to a new location. A block of text may consist only of a single character or as many as several pages.

To illustrate the use of MOVE/COPY let's make up a new file consisting of an address list. If you wish to save HAMLET, do so now using:

```
   STORE/GET
 V:SAVE WK-SPACE
III:DRIVE A
VI:SAVE WK-SPACE
```

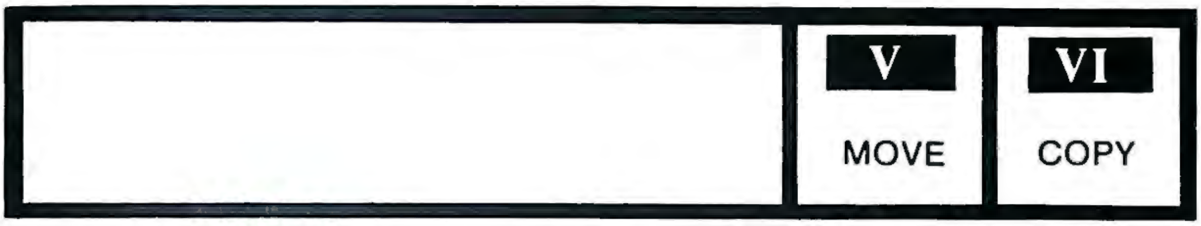
Clear the workspace with:

```
   CLEAR
VI:CLEAR WK-SPACE
VI:FINAL CLEAR
```

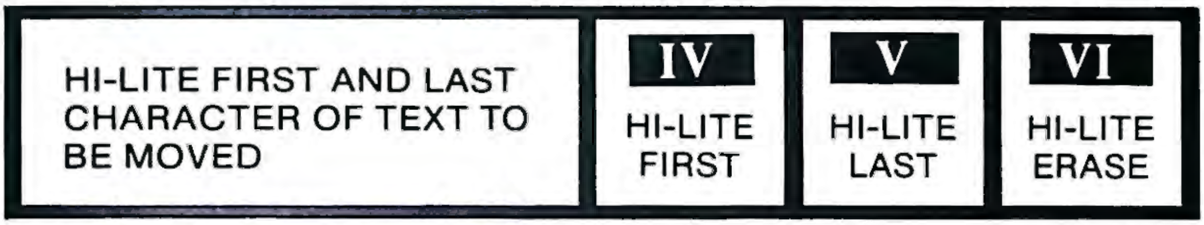
Enter the following text:

```
Mr. A Baker◀
3030 Dune Dr.◀
Reno, NV 34567◀
◀
Mr. R. Able◀
2020 Tundra Dr.◀
Anchorage, AK 23456◀
◀
Mr. B. Conte◀
1010 Sundrench Ave.◀
Miami, FL 12345◀
```

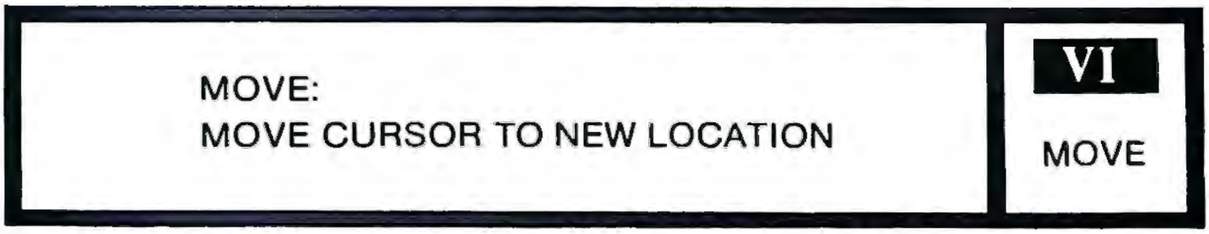
This text could be the beginning of a large mailing list. If it were, it would probably be most convenient to place the names in alphabetical order. To do so, first press the MOVE/COPY key. The smart keys will change to:



Press key V (MOVE), which changes the keys again:


HI-LITE FIRST AND LAST CHARACTER OF TEXT TO BE MOVED

Position the cursor beneath the "M" in Mr. R. Able. Then, press key IV (HI-LITE FIRST). Now, move the cursor to the ◀ symbol located below the "A" in Anchorage and press key V (HI-LITE LAST). The text will momentarily be underlined, then disappear. The smart keys then change to:


MOVE:
MOVE CURSOR TO NEW LOCATION

Move the cursor to the "M" in Mr. A. Baker then press key VI (MOVE). The text should appear as shown below:

```
Mr. R. Able◄
2020 Tundra Dr.◄
Anchorage, AK 23456◄
◄
Mr. A. Baker◄
3030 Dune Dr.◄
Reno, NV 34567◄
◄
Mr. B. Conte◄
1010 Sundrench Ave.◄
Miami, FL 12345◄
◄
```

Store this file with the name ADDRESS. We will access this file later as we continue our mailing list example.

SAVING HIGHLIGHTED TEXT TO A FILE

It is often useful to designate sections of a document for special operations. SmartWriter uses a procedure called highlighting to perform this function. The HI-LITE mode is used to specify a section of a document to be printed, saved, or relocated within a file.

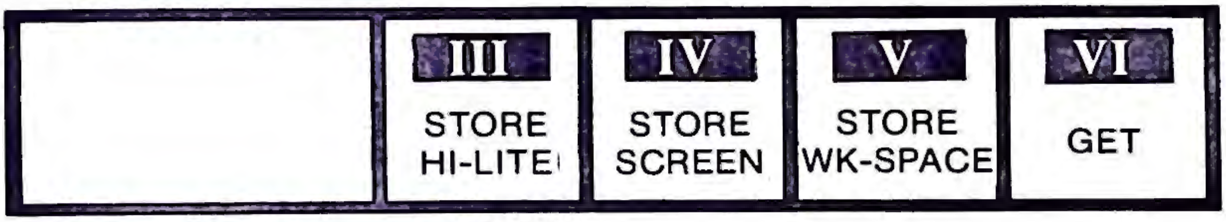
For example, consider the situation in which a standard form letter is to be customized for each recipient. This procedure can be performed quickly and easily through the use of the HI-LITE feature.

The HI-LITE mode can be selected with the smart key labeled IV:HI-LITE. When this selection is chosen, the label will change from HI-LITE to HI-LITE OFF. This allows the same key to be used to cancel the HI-LITE mode. When the HI-LITE mode is in effect, any typed characters will be underlined. Characters can also be underlined through the use of the cursor control keys. For example, if the cursor originally appears at the beginning of a line, the → key can be used to underline the text on that line.

A similar procedure is used to remove the underlining. The smart key labeled V:HI-LITE ERASE allows the underlining to be removed. When this mode is selected, the cursor can be used to remove the underlining from a character.

Practice using the HI-LITE feature by loading the ADDRESS file into the workspace. Move the cursor to the beginning of the line that contains the words Mr. A. Baker. Press the smart key labeled IV:HI-LITE. Proceed by using the → key to move the cursor to the end of the line. The entire line should now be underlined. Use the cursor control keys to underline the remainder of Mr. Baker's address.

Now that Mr. Baker's address has been underlined, let's save it on a file. First, press STORE/GET. The smart keys will respond with:

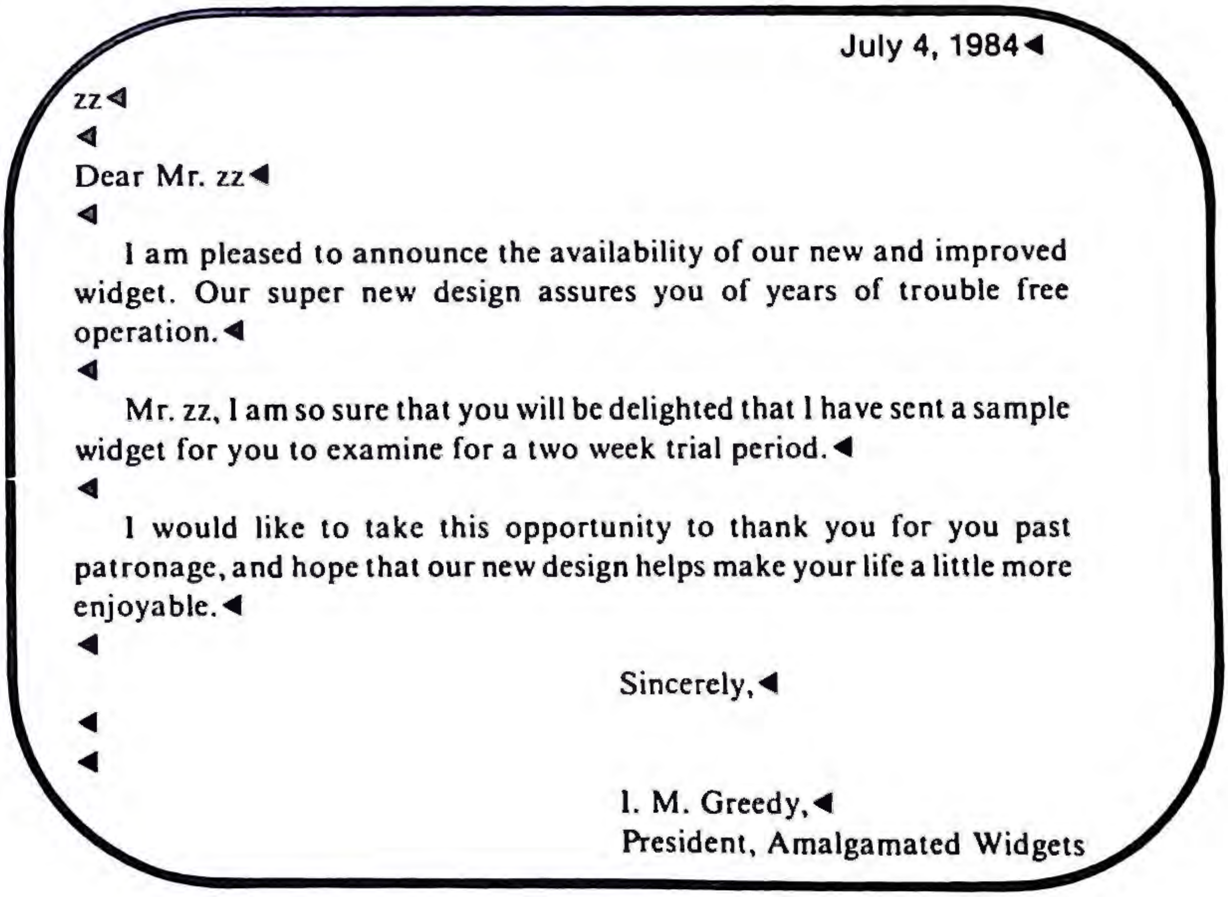


	III	IV	V	VI
	STORE HI-LITE	STORE SCREEN	STORE WK-SPACE	GET

Enter III (STORE HI-LITE). After responding to the drive prompt, enter TEMP as the name of the temporary file. Wait a few moments for the file to be written to the Digital Data Pack.

Example: A Form Letter

CLEAR the workspace with the CLEAR/VI:CLEAR WK-SPACE/-VI:FINAL CLEAR sequence. Copy the following text:



July 4, 1984◄

zz◄

◄

Dear Mr. zz◄

◄

I am pleased to announce the availability of our new and improved widget. Our super new design assures you of years of trouble free operation.◄

◄

Mr. zz, I am so sure that you will be delighted that I have sent a sample widget for you to examine for a two week trial period.◄

◄

I would like to take this opportunity to thank you for you past patronage, and hope that our new design helps make your life a little more enjoyable.◄

◄

Sincerely,◄

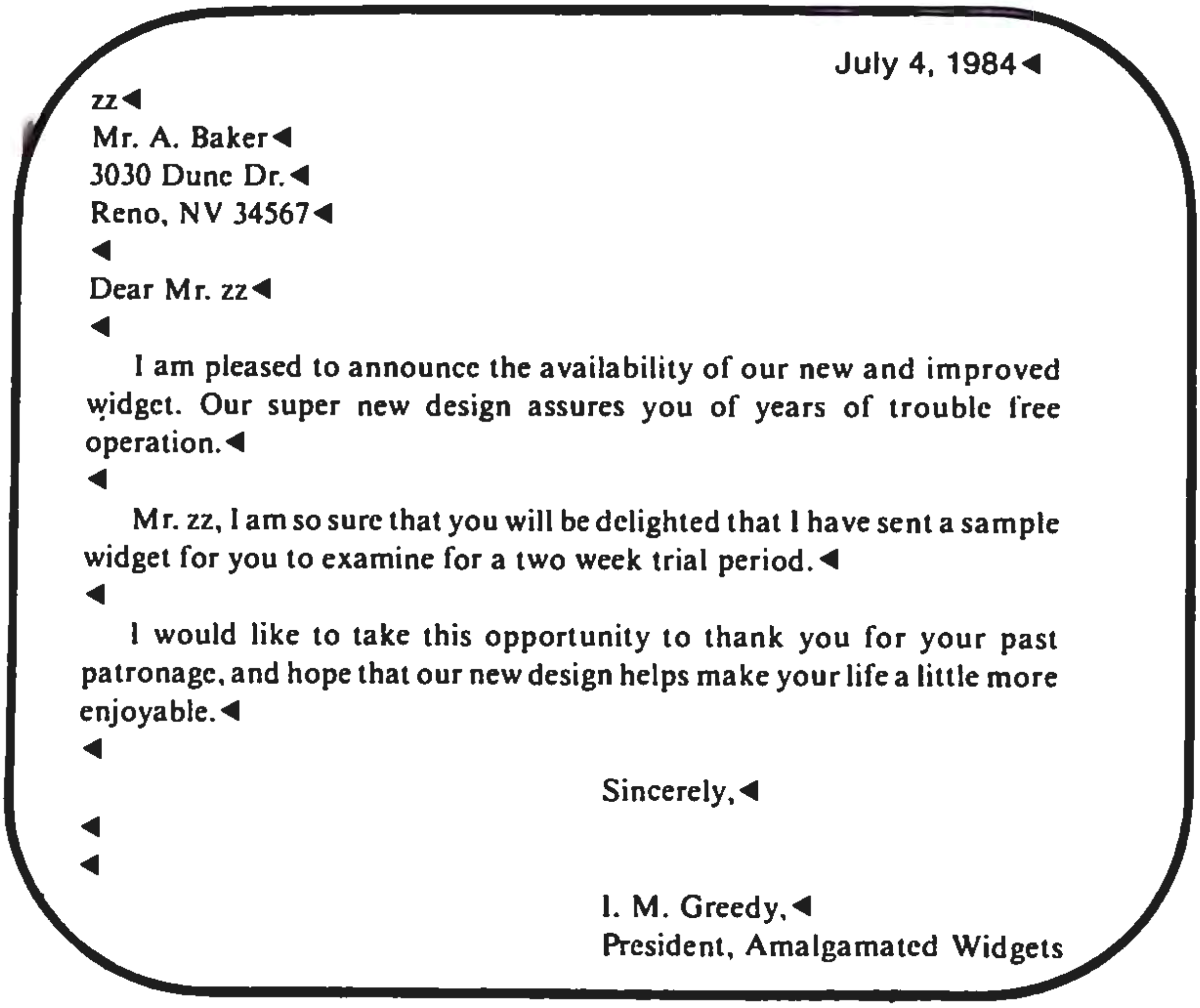
◄

◄

I. M. Greedy,◄
President, Amalgamated Widgets

Save this text under the name FORMLET with STORE/GET/V:SAVE WK-SPACE/III:DRIVE A/VI:SAVE WK-SPACE. Notice that we never used names in writing FORMLET, only the letters "zz".

Position the cursor to the beginning of the file with the HOME key. Press key III (SEARCH) and answer "zz" to the prompt of text to search for. Complete the command sequence by pressing the smart key labeled VI:START SEARCH followed by VI:DONE. Press STORE/GET/-VI:GET/III:DRIVE A and move the cursor to the file TEMP you created earlier. Press key VI (GET FILE). The Digital Data Pack will require about 15 seconds to locate the file and bring it into the workspace. The following display will appear:



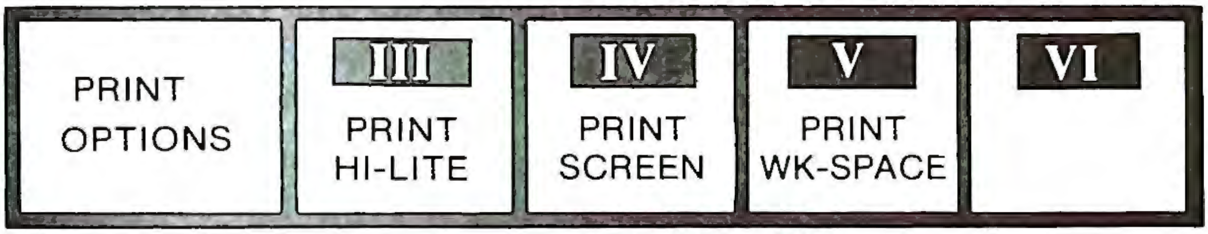
```
                                                  July 4, 1984◀
zz◀
Mr. A. Baker◀
3030 Dune Dr.◀
Reno, NV 34567◀
◀
Dear Mr. zz◀
◀
    I am pleased to announce the availability of our new and improved
widget. Our super new design assures you of years of trouble free
operation.◀
◀
    Mr. zz, I am so sure that you will be delighted that I have sent a sample
widget for you to examine for a two week trial period.◀
◀
    I would like to take this opportunity to thank you for your past
patronage, and hope that our new design helps make your life a little more
enjoyable.◀
◀
                              Sincerely,◀
◀
◀
                              I. M. Greedy,◀
                              President, Amalgamated Widgets
```

First, use the cursor keys and backspace to remove the line "zz◀". Then, to finish this letter, replace the next two occurrences of "zz" with the name "Baker". *One* way of doing this (by now you can probably think of three other ways) is to press III:SEARCH/VI:START SEARCH/-V:REPLACE/BAKER/VI:REPLACE ALL. The two "zz" occurrences should now be replaced with "Baker". Notice that it was not necessary to re-enter the pattern "zz" to SEARCH. That is an old pattern will remain in order within the SEARCH buffer until it has been replaced.

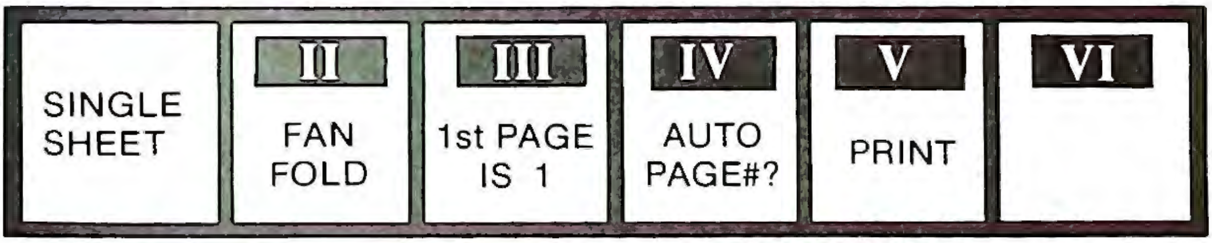
To save this workspace, remember that the last file accessed was TEMP. If a new name is not entered in response to the STORE sequence prompt, then this file will be saved as TEMP. Save the file as BAKER.

Printing Text

Outputting text to the printer using SmartWriter is a simple process. To illustrate this, bring the file named BAKER into the workspace. Pressing the PRINT key changes the smart keys to:



Pressing key V (PRINT WK-SPACE) changes the keys again:



Before printing, insert a sheet of paper and center it on the roller. The amount of paper protruding above the print head should equal the top vertical margin (explained later). Since the default top margin is set to 6 lines (or 1 inch), this amount of paper should protrude above the print head. If the top vertical margin is later changed (as explained later), the paper must be advanced appropriately.

When the V key is pressed, the workspace contents will be output to a standard 8½ x 11 inch sheet of paper. After a page has been filled, the printer will pause so as to enable the operator to insert a new sheet. After this sheet is in position, printing can be resumed by pressing the V key (PRINT).

If a lengthy file is being output, you may wish to use **fanfold paper** in order to avoid having to insert a new sheet of paper for each page. Fanfold paper is a roll of paper which has been folded into a standard 8½ x 11 inch size and is perforated so that it can be easily separated. Press key II when fanfold paper is being used. When this option is selected, SmartWriter will automatically advance from one page to the next using the vertical margin specified.

SmartWriter allows for automatic page numbering with key IV (AUTO PAGE #). When this key is pressed, pages will be numbered in their upper right hand corner. Key II (FIRST PAGE IS 1) can be used to begin page numbering with a page other than 1. For example, to begin numbering with page 5, press key III four times. Notice that the auto-repeat feature does not work with this key.

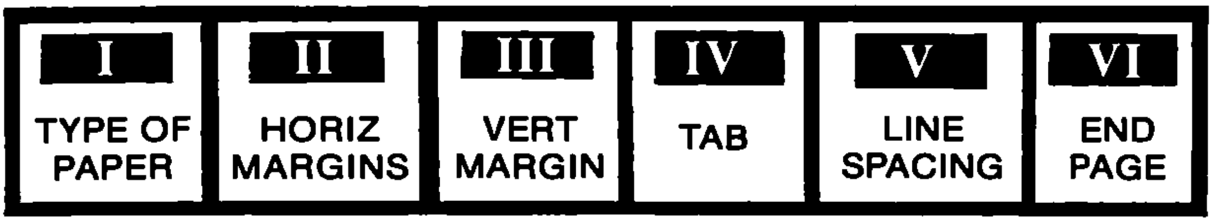
If a printout must be stopped for some reason, press key V (STOP PRINT). Printing can be resumed by pressing this key again.

Formatting Text

So far, the text that we have prepared has been of standard format. By this we mean that it is single spaced on 8½ x 11 inch typing paper with standard margins. SmartWriter allows these defaults to be changed before or after the text has been written.

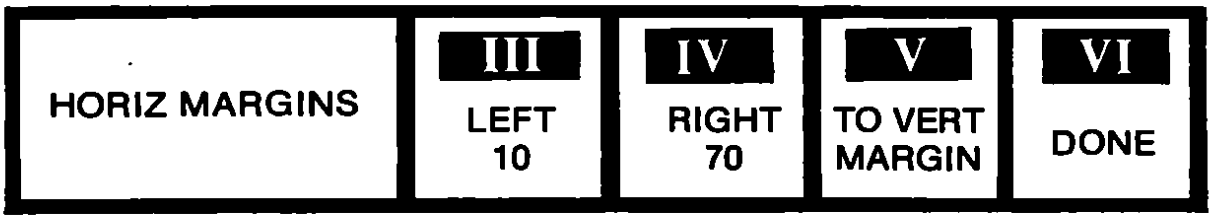
MARGINS

SmartWriter has default horizontal margins set at 10 and 70. As mentioned previously, these are visible as red marks on the horizontal margin scale. To change the left margin, first press key I (MARGIN/-TAB/ETC). The following labels will appear:



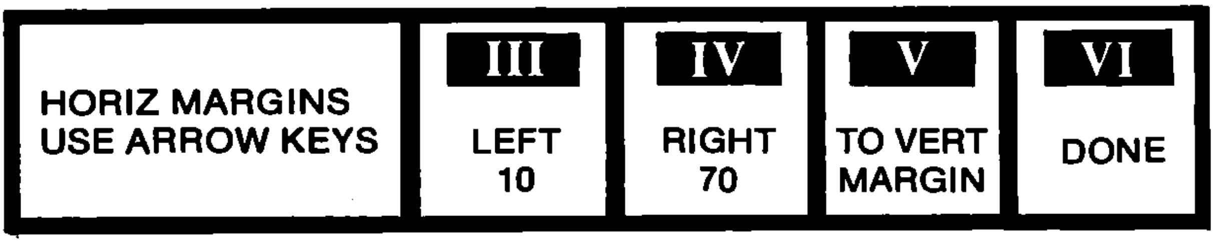
I	II	III	IV	V	VI
TYPE OF PAPER	HORIZ MARGINS	VERT MARGIN	TAB	LINE SPACING	END PAGE

Next, press key II (HORIZ MARGIN). The following labels will appear:



	III	IV	V	VI
HORIZ MARGINS	LEFT 10	RIGHT 70	TO VERT MARGIN	DONE

To change the left margin, first press key III. Notice that the smart key labels do not change, although the message at the left does:



	III	IV	V	VI
HORIZ MARGINS USE ARROW KEYS	LEFT 10	RIGHT 70	TO VERT MARGIN	DONE

The → and ← keys may now be used to adjust the left margin. Pressing the ← key moves the margin to the right, while pressing the → key moves the margin to the left. Pressing key IV allows the right margin to be moved in the same manner. The current value of the left margin is displayed in label box III. The value of the right margin is displayed in label box IV.

Use the cursor keys to change the left margin to 40. Let's use this setting to address an envelope. First, clear the workspace with CLEAR/-VI:CLEAR WK SPACE/VI:FINAL CLEAR. Next, return the file named TEMP (which should still contain Mr. Baker's address) with STORE/GET/VI:GET/III:DRIVE A/TEMP/VI:GET FILE.

Examine the horizontal margin scale. Notice that the horizontal margin markers have been reset to their default position. When a file is retrieved and brought into the workspace, its format setting will also be retrieved as well. This point is important to remember when working with files. It is a good idea to always recheck the horizontal and vertical margin markers when a file is read into the workspace.

We can now print the address on the envelope. Change the left margin with I:MARGIN/TAB/ETC/II:HORIZONTAL MARGIN/III:LEFT 10/.../III:LEFT 40/VI:DONE. Next, insert an envelope into the printer and print the address using PRINT/V:PRINT WK SPACE/V:PRINT.

If the workspace was cleared, remember that the formats just set will still be in effect. If a new file is to be set with standard defaults, either the left margin must be reset or the RESET COMPUTER switch must be pressed followed by ESCAPE/WP.

Margin settings can be of value in the moving window format in that they can be set so as to allow all of the text to be visible without using the cursor keys. Change to the moving window format with II:SCREEN OPTIONS/VI:MOVING WINDOW. Since each line holds 36 characters, one way of seeing them all is to set the right margin to 46 with I:MARGIN/TAB/ETC/II:HORIZ MARGIN/IV:RIGHT 70/.../IV:RIGHT 46/VI:DONE.

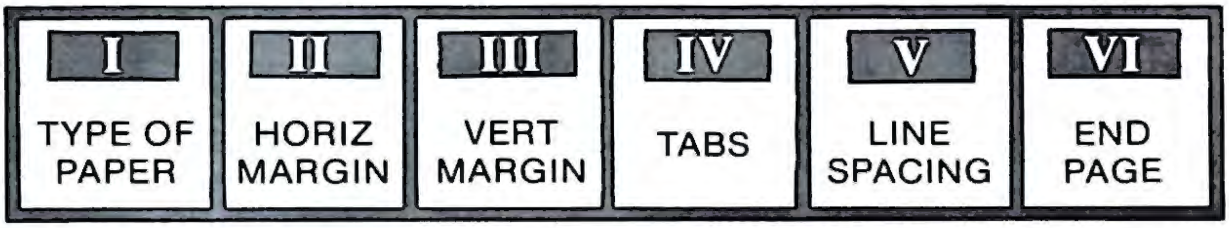
Once the margin has been set, clear the workspace and enter the following:

```
Four score and seven years ago our
fathers brought forth on this
continent a new nation conceived in
liberty and dedicated to the
proposition that all men are created
equal. Now we are engaged in a great
civil war testing whether that nation
or any other nation so conceived or
dedicated can long endure.
We are met on a great battlefield of that
war. We have come to dedicate a
portion of that field as a final
resting place for those who here gave
their lives that this nation may
live. It is altogether fitting and
proper that we should do this
```

Now return the right margin to its default value with I:MARGIN/TAB/ETC/II:HORIZONTAL MARGIN/IV:RIGHT 46/.../IV:-

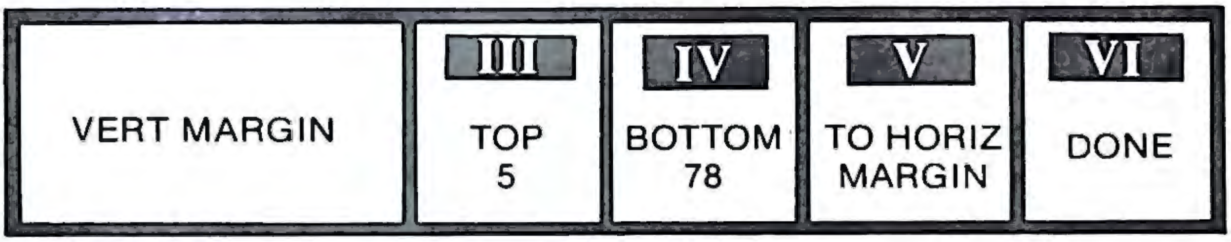
RIGHT 70/VI:DONE. The entire workspace was automatically **reshaped** to the new margin.

The vertical margins may also be changed with the MARGIN/TAB/ETC key. The standard margin setting are 6 lines at the top and bottom. This allows approximately one inch of clearance. To set the top margin at 9 lines (about 1½ inches), begin by pressing key I (MARGINS/TAB/ETC):



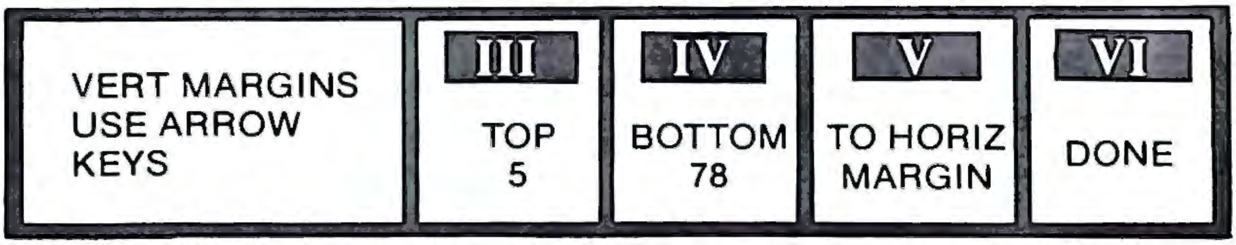
I	II	III	IV	V	VI
TYPE OF PAPER	HORIZ MARGIN	VERT MARGIN	TABS	LINE SPACING	END PAGE

Next, press key III (VERT MARGIN). The following will appear:



	III	IV	V	VI
VERT MARGIN	TOP 5	BOTTOM 78	TO HORIZ MARGIN	DONE

To change the top margin, press key III. Notice that the message at the left changes:



	III	IV	V	VI
VERT MARGINS USE ARROW KEYS	TOP 5	BOTTOM 78	TO HORIZ MARGIN	DONE

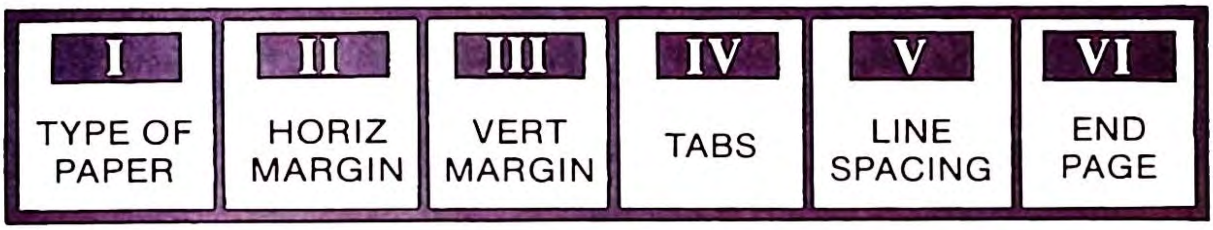
Using the ↑ and ↓ keys, change the top margin to 9. These keys are used in a manner similar to the usage of the → and ← keys with the horizontal margin. To change the bottom margin, first press key IV. Then, use the ↑ and ↓ keys.

ENDING A PAGE

It is often convenient to end a page before it has been completely filled with text. One means of doing so would be to continue pressing RETURN until the vertical cursor marker had traveled from the bottom of the scale to the top. SmartWriter makes the job a bit easier with the END PAGE command. We'll use an example to illustrate END PAGE usage. First, CLEAR the workspace and enter the following text:

```
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
◄
                 How I Spent My Summer Vacation ▤
```

Don't bother to count the ◀'s, just press the RETURN key until the vertical cursor marker has descended to the half-way hash mark. To end the page, first press key I (MARGIN/TAB/ETC). This will generate the following labels:



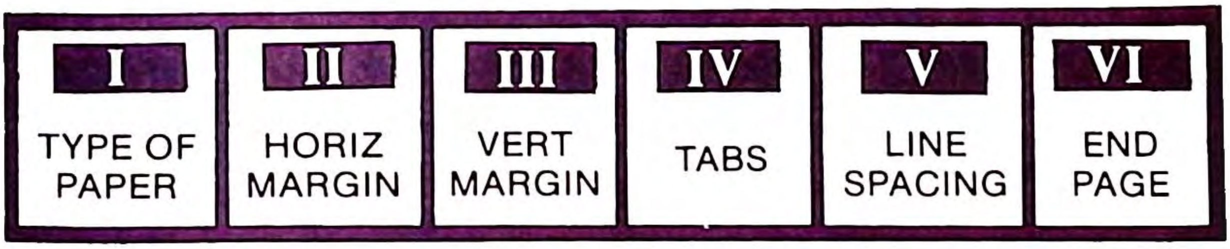
Press key VI (END PAGE). An end page marker, ▤ , will appear after "Vacation". This marker indicates that the page will end here. Move to the next line using the cursor keys and enter the following:

```
After climbing Mt. Everest and
swimming the English Channel, I
was about to cross the Sahara on a
special reconnaissance mission
when I stopped to ask myself
. . . why am I doing all this? . . .
```

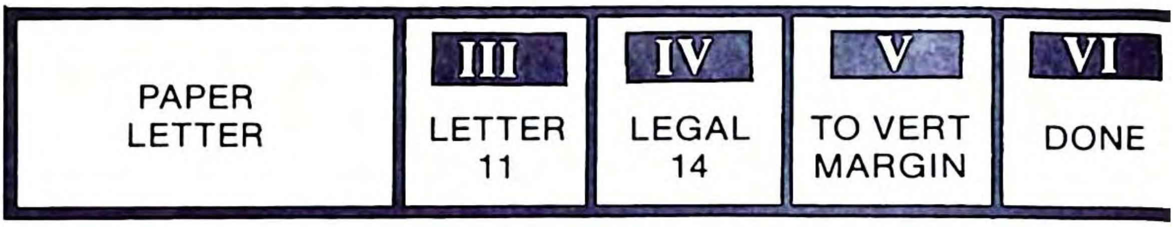
Now, print the workspace with PRINT/V:PRINT WK SPACE/-V:PRINT. After the title (How I spent . . .) has been output, the printer will pause. At this point, insert a new piece of paper and press key V (V:PRINT).

CHANGING PAPER SIZE

Although the standard 11 inch format is suitable for most applications, SmartWriter allows this default value to be changed to legal size, which is 14 inches. To do so, first press key I (MARGIN/TAB/ETC):



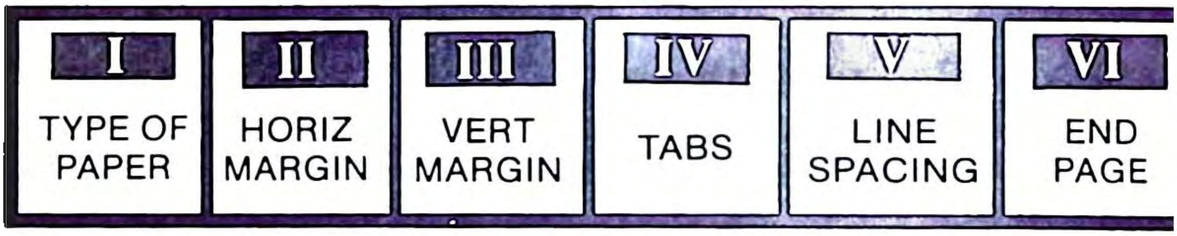
Pressing key I (TYPE OF PAPER) changes the labels to:



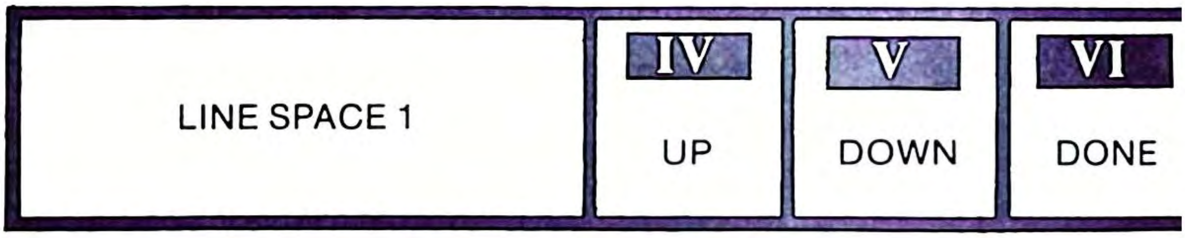
Pressing key IV (LEGAL 14) causes the message at the left of th smart key labels to change from "PAPER LETTER" to "PAPE LEGAL". After pressing key VI (VI:DONE), a "14" will be available a the top of the vertical margin scale. Legal size paper can now be used wit ADAM's printer.

CHANGING LINE SPACING

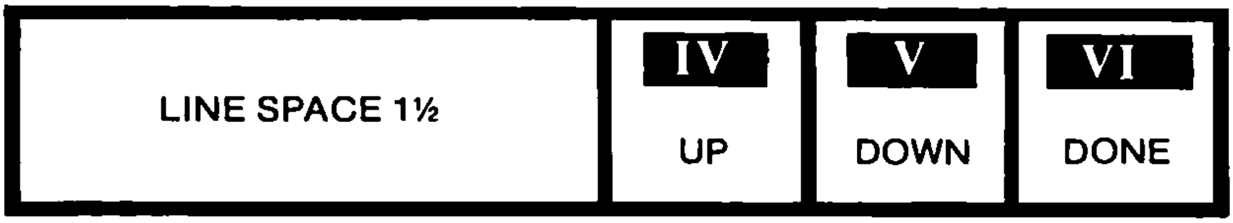
So far all of our text has been printed using single spacing. Smart /riter allows up to 84 lines of spacing with 1/2 space intervals. To alte spacing, begin by pressing key I (MARGIN/TAB/ETC):



Press key V (V:LINE SPACING). The smart key labels will change to:



To increase line spacing, simply press key IV (UP). The label at the left will reflect a 1/2 line increase:



LINE SPACE 1½

IV UP

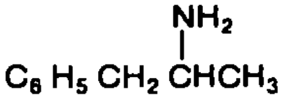
V DOWN

VI DONE

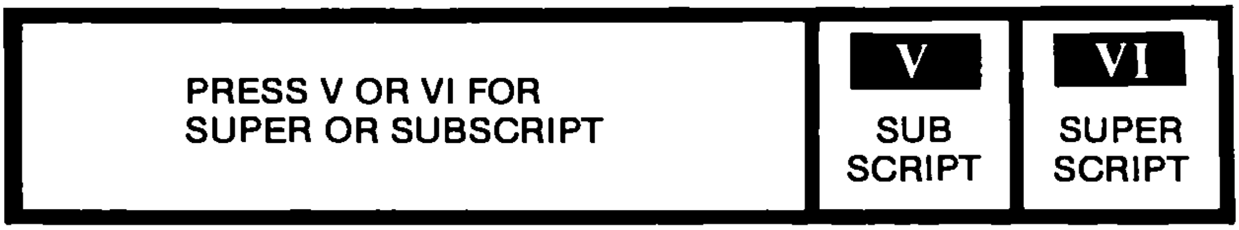
Likewise, pressing key V (DOWN) will decrement the line spacing by 1/2.

SUPER— AND SUBSCRIPTING

Suppose you are writing a chemistry report and wish to show the chemical formula for 2 amino-1 phenylpropane:



This can be easily output with SmartWriter. Enter C followed by key VI (SUPER/SUBSCRIPT). The smart key labels will change as follows:

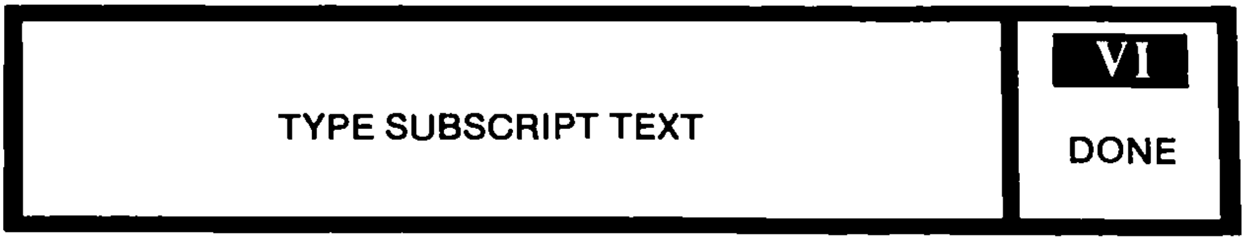


PRESS V OR VI FOR
SUPER OR SUBSCRIPT

V SUB SCRIPT

VI SUPER SCRIPT

Press key V (SUBSCRIPT):



TYPE SUBSCRIPT TEXT

VI DONE

Type in the 6 and then press key VI (DONE). Notice that the special characters "└ " and " ┘" surround the subscripted text. Finish entering the normal and subscripted text and then center it using INSERT. The SUPER/SUBSCRIPT feature only works if the line spacing had been set to 1½ or more. Final text should resemble that shown below:

NH└2┘◄

|◄

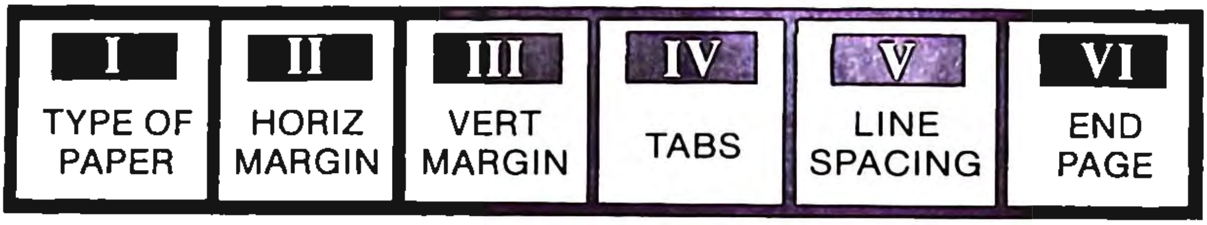
C└6┘H└5┘CH└2┘CHCH└3┘

Superscripts are written in the same manner, except that the special characters which surround the text are "┌ " and " ┐".

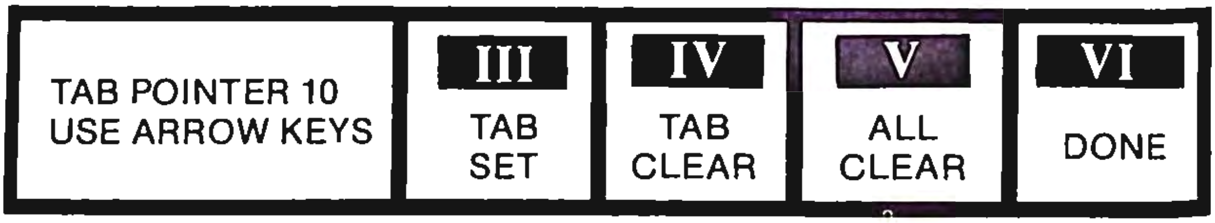
TABS

SmartWriter makes table preparation a cinch. As mentioned earlier, the moving window is appropriate for this kind of work, so change to this format now.

Suppose that we wanted to create a computerized calendar. One way of accomplishing this would be to make a 7 x 5 grid and fill in the numbers for any particular month. Tabs would be a great help in constructing these grids. The default tabs are set every 5 spaces, but these are easily changed. To see this, press key I (MARGIN/TAB/ETC):

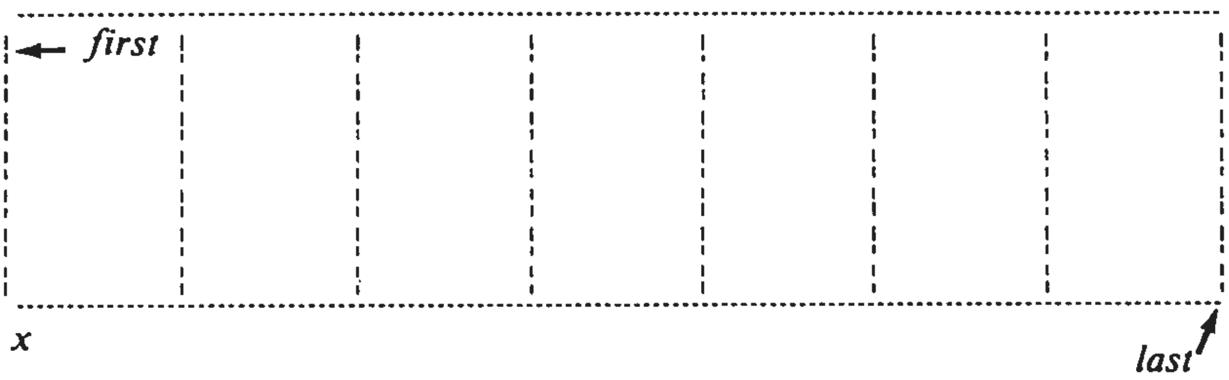


Press key IV (TABS):


TAB POINTER 10
USE ARROW KEYS

Cursor movement with the arrow keys is reflected in the message to the left of the smart key labels. The left hand margin should be set at 10 and the right at 70 unless these had been changed using the MARGIN command. The default tabs, which are set every 5 spaces, are visible as white dots on the horizontal margin scale. A tab can be set by simply moving to the desired position and pressing key III (TAB SET). Errors can be cleared by moving the cursor to the unwanted tab and pressing key IV (TAB CLEAR). For the purpose of our example, enter 10,18,26, 34,42,50,58, and 66. Fix these settings by pressing key VI (DONE). Enter the following lines using the tab to position the cursor:

◀
◀
◀



This grid could be finished by hand, but it would be easier to use the MOVE/COPY command to copy a repeating portion. HI-LITE the characters circled as "first" and "last". Then, bring the cursor to the location indicated by "*x*". COPY this block of text four times, always remembering to return the cursor to the southwest corner of the text.

To add day labels, bring the cursor to the RETURN marker (◀) just above the northwest corner of the calendar. Use the space bar to "push" the marker to the center of the first column and type "S" for Sunday. Do the same for the rest of the days of the week. The top of your file should resemble the following:

```
◄
◄
◄
------------------------------------------------------------------------------
     S          M          T          W          T          F          S◄
------------------------------------------------------------------------------
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
```

Use the TAB key and the ← cursor key to position the cursor while entering the dates. Short reminders can be written in the blocks. If an incorrect entry was made, it is probably easier to use the cursor keys and overtype than it is to INSERT the text vertical lines. The final product should resemble that depicted below:

```
◄
◄
◄
------------------------------------------------------------------------------
     S          M          T          W          T          F          S◄
------------------------------------------------------------------------------
|          |          |          |                    1|         2|         3|
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
------------------------------------------------------------------------------
|         4|         5|         6|         7|         8|         9|        10|
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
------------------------------------------------------------------------------
|        11|        12|        13|        14|        15|        16|        17|
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
------------------------------------------------------------------------------
|        18|        19| 20       |        21|        22|        23|        24|
|          |          |          |          | Schools  |          |          |
|          |          |          |          |   Out    |          |          |
|          |          |          |          |          |          |          |
------------------------------------------------------------------------------
|        25|        26|        27|        28|        29|        30|        31|
| Christmas|          |          |          |          |          | New Years|
|          |          |          |          |          |          | Eve Party|
------------------------------------------------------------------------------
```

Overview: The Smart Keys Revisted

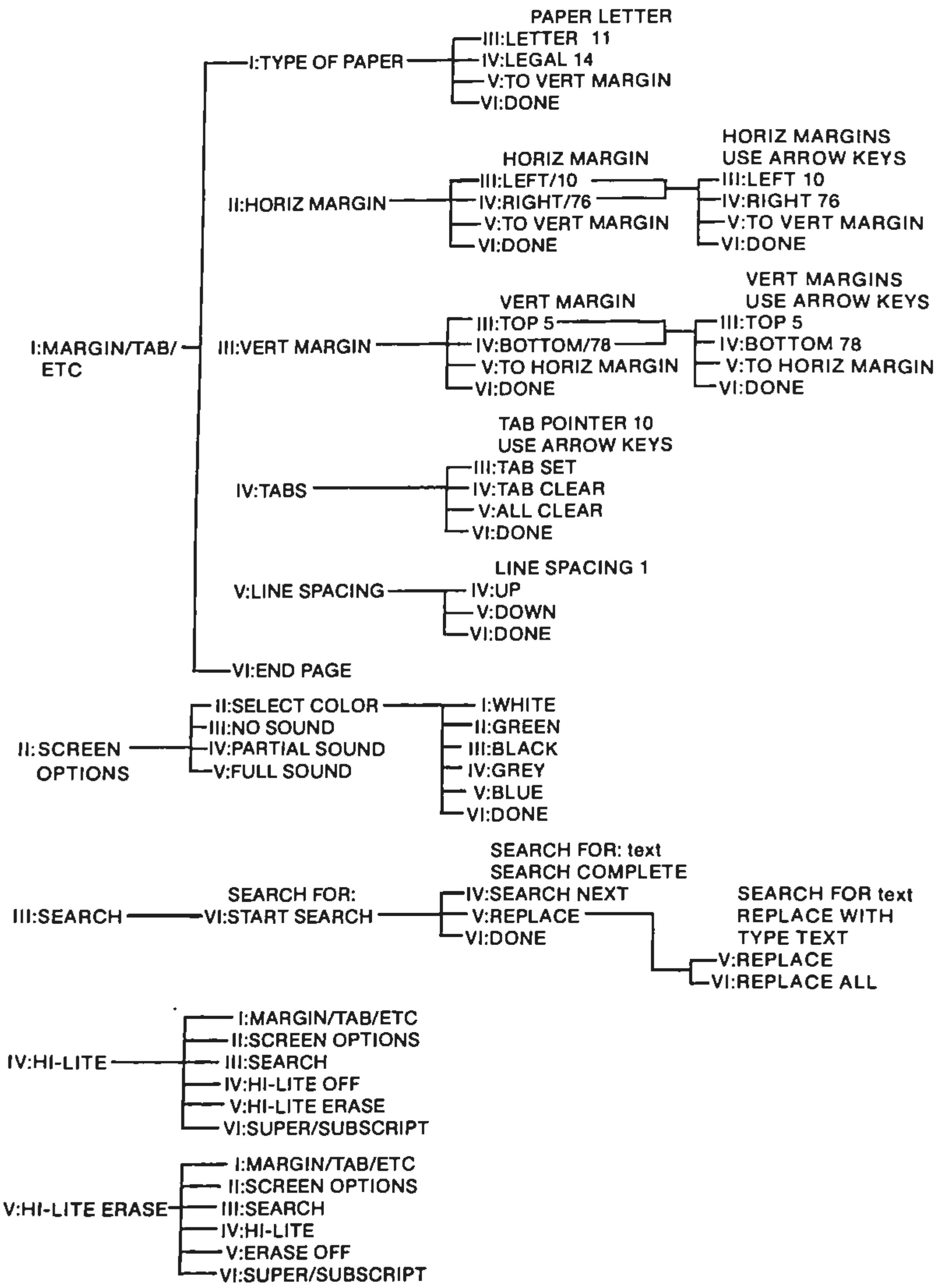
By now we hope that you feel comfortable using SmartWriter. Although the command entry sequence might seem confusing at first, it is actually quite simple.

When entering a command sequence, the entry proceeds from a general command such as SCREEN OPTIONS to a very specific one, such as setting the screen color to WHITE. This process can be envisioned as a journey up the branches of a tree, from the broad subject of SCREEN OPTIONS to the narrow specification of screen color to WHITE.

Figure 3-8 depicted the **tree structure** of the standard menu. As you can see, any command can be executed with at most four keystrokes. Since this information is displayed during the command entry process, there is no need to memorize the chart. One option not listed in Figure 3-8 is EXCAPE/WP. This option, which is available at every node of the tree, returns SmartWriter to the "trunk", or standard menu.

Figures 3-9 through 3-14 show the structures of the MOVE/COPY STORE/GET, CLEAR, INSERT, PRINT, and DELETE commands respectively. We hope that these will serve as a quick reference guide to the commands introduced in this chapter.

FIGURE 3-8. Standard Menu Command Structure



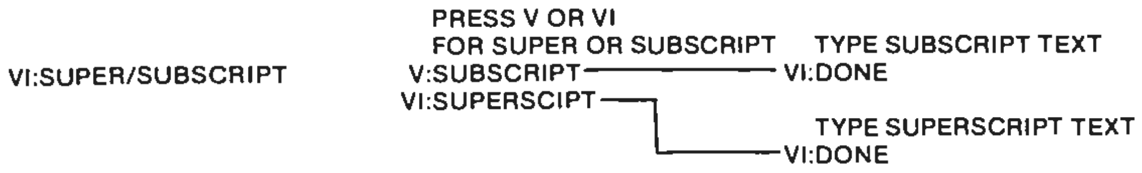


FIGURE 3-9. MOVE/COPY Command Structure

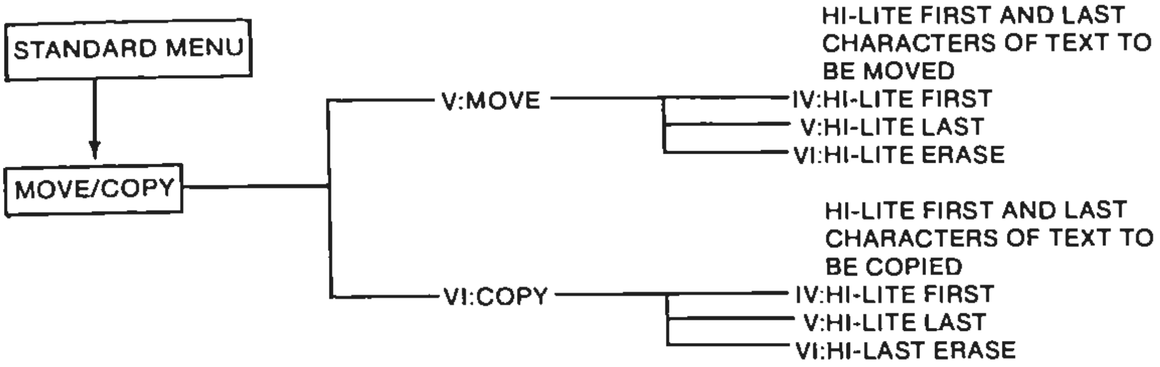


FIGURE 3-10. STORE/GET Command Structure

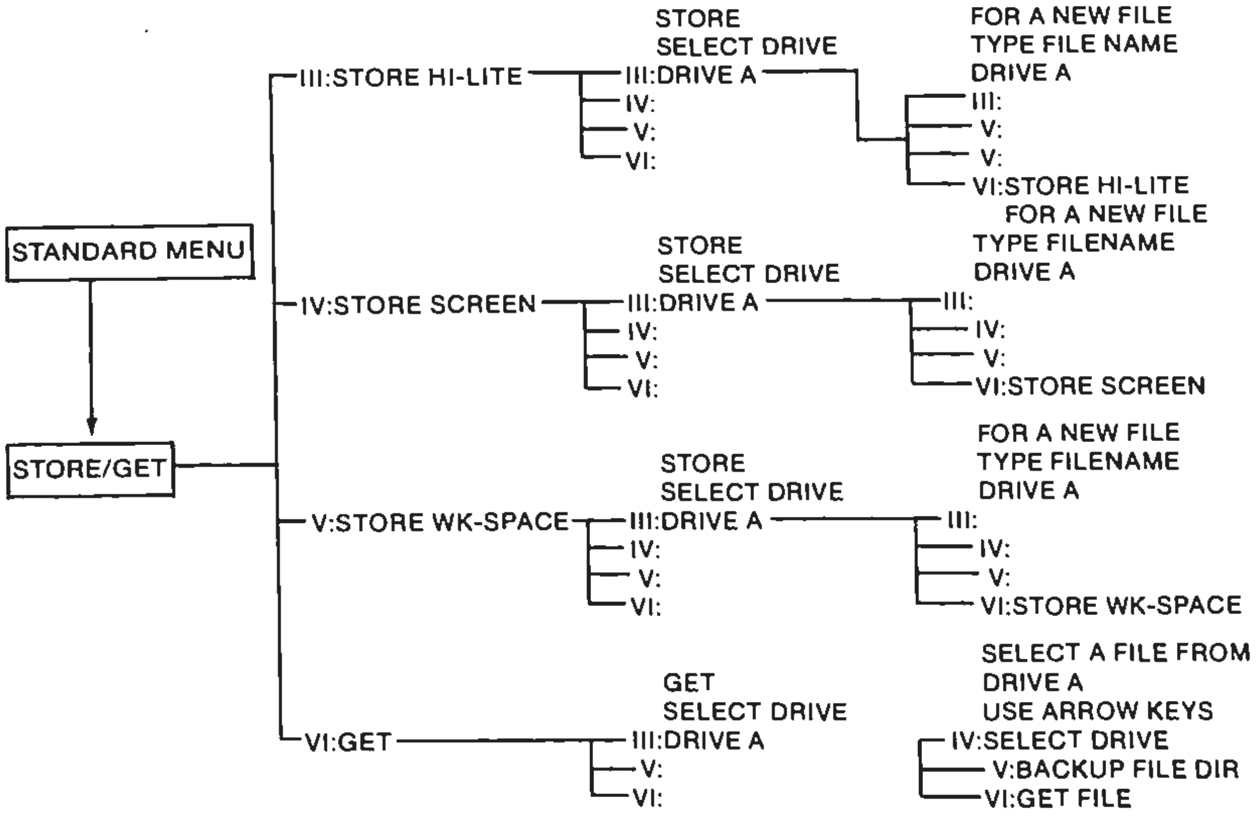


FIGURE 3-11. CLEAR Command Structure

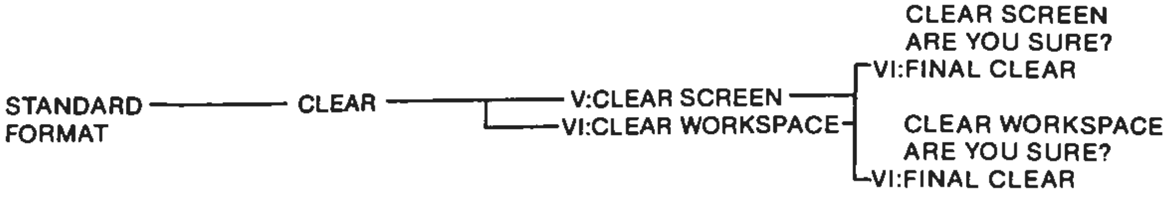


FIGURE 3-12. INSERT Command Structure

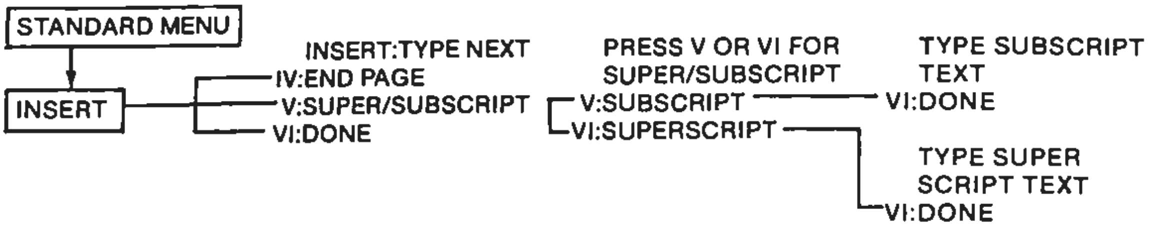


FIGURE 3-13. PRINT Command Structure

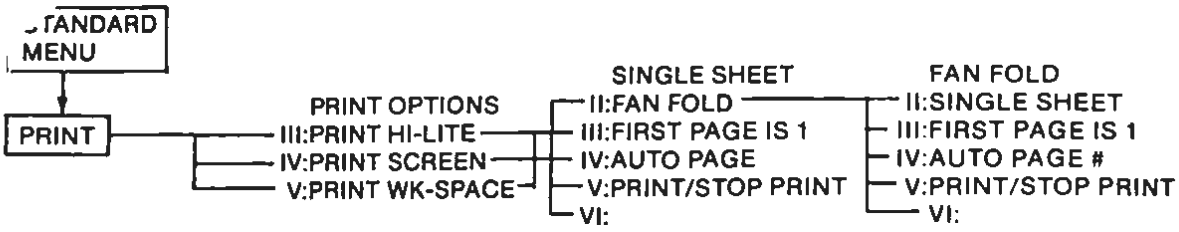
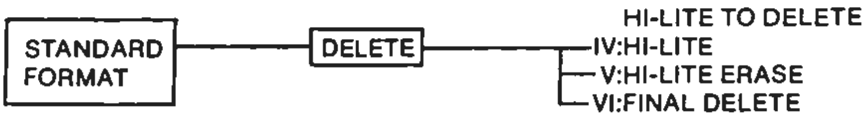


FIGURE 3-14. DELETE Command Structure



4

Introduction to SmartBASIC

Introduction

In this chapter, we will provide an overview of the background and history of SmartBASIC as well as the operating details you will need to know to begin using SmartBASIC. These include start-up, program entry, statement structure, program editing, program saving, and program loading.

BASIC PROGRAMMING LANGUAGE BACKGROUND

BASIC is probably the most widely used programming language for microcomputers, with the ADAM being no exception.

BASIC was developed at Dartmouth College in the early 1960's by professors John G. Kemeny and Thomas E. Kurtz. BASIC was designed as a simple, easy-to-use programming language.

There are many versions of BASIC used with various computers. As a result, the version of BASIC that is used with the ADAM is not the same as the version of BASIC used with other computers. However, the fundamentals of the BASIC language are the same, regardless of the version.

PROGRAMMING LANGUAGES

BASIC, is a **high level programming language.** With a high-level language, the programmer need not have a knowledge of the **machine language** used by the computer's microprocessor. With machine or **assembly language,** an in-depth understanding of the computer and microprocessor is required to write programs.

With a high-level language such as BASIC, commands are generally specified in English words that can be associated with the operation to be performed. For example, the BASIC command PRINT instructs the computer to display information. As a result, it is usually much easier to program in a high-level language such as SmartBASIC than in a machine or assembly language.

COMPILED vs. INTERPRETED LANGUAGES

High-level computer languages are often distinguished as being ,ither **compiled** or **interpreted** languages.

A compiled language program consists of the **source code** and the **compiled code**. The source code consists of the program statements in their original form. For example, the following is a line of source code from a program written in the CBASIC compiled language:

```
100 INPUT "ENTER TODAY'S DATE:";DATE.1
```

The source code is processed by a program known as a **compiler** into the compiled code. The compiled code is very similar to the machine language used by the microprocessor. The compiled code is the code actually used when a compiled program is run. A program known as a **run-time monitor** is used to run the compiled program.

An interpreted language consists of only the source code. The source code is translated line-by-line directly into machine language instructions. SmartBASIC is an interpreted language.

One advantage of interpreted languages over compiled languages is that interpreted language programs are more easily developed. When working with interpreted languages, a programmer need only write a program, enter it, run it, and alter it at his own leisure. When working with a compiled language, the source code must be recompiled every time it is edited. This can be frustrating during the program debugging process.

One advantage of compiled languages over interpreted languages is that the execution time is much faster. The compiled code is much closer to the machine language than the source code. Since interpretation is not necessary, execution of compiled code is much faster.

GETTING STARTED WITH SMARTBASIC

As we discussed earlier, BASIC is a high-level language which must be interpreted into the microprocessor's native machine language. This is accomplished with a program known as an **interpreter.**

Before a BASIC program can be executed, the SmartBASIC interpreter must be loaded into the computer's memory. The Digital Data Pack that contains the BASIC interpreter is labeled SmartBASIC. Begin by inserting the Data Pack into the port in the console as shown in Figure 4-1.

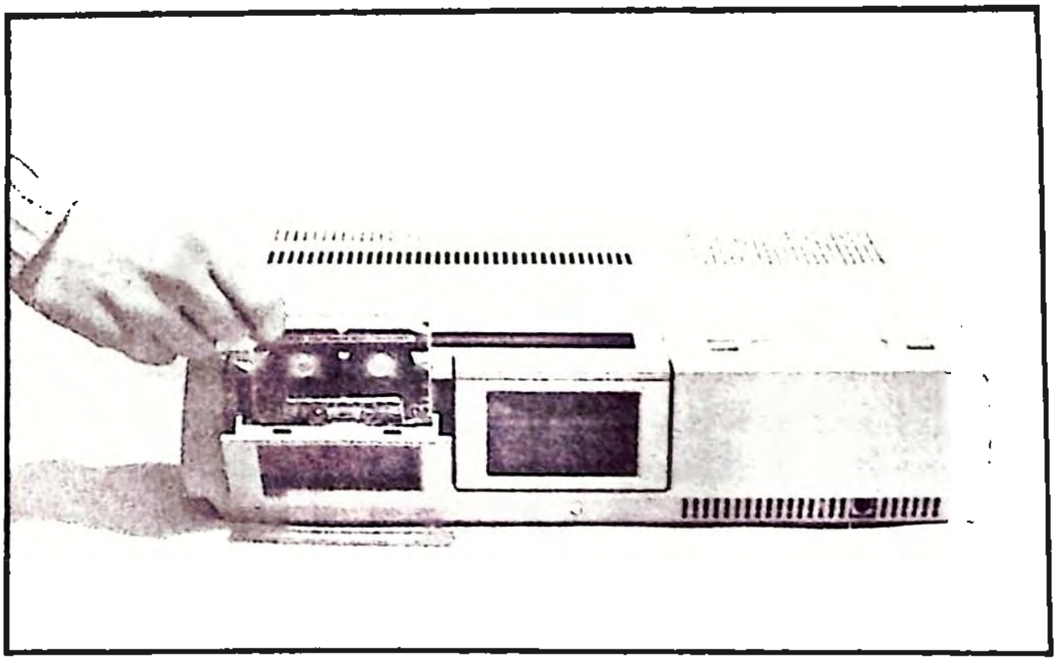


FIGURE 4-1. Inserting the Digital Data Pack

Once the Data Pack has been inserted in the drive, the loading procedure is ready to begin. A switch labeled COMPUTER RESET is located on top of the main computer console. Slide this switch forward and release it. The Digital Data Pack Drive will operate for about a minute before the programming language will be ready to use.

The following message will appear at the top of the display when the loading procedure has been completed.

```
Coleco SmartBASIC V1.0
```

At the bottom of the display, two special characters appear. The first character (]) is called the **prompt.** This character will be displayed whenever the computer is ready to accept a command or statement.

The second character is a blinking underline (__). This character is known as the **cursor.** The cursor indicates the position on the display where the next character entered via the keyboard will appear.

IMMEDIATE AND PROGRAM MODES

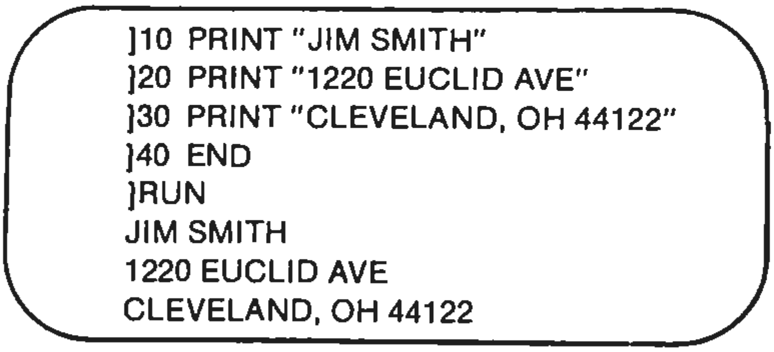
The immediate mode is also known as the direct or calculator mode. In the immediate mode, most BASIC command entries result in the instructions being executed without delay. For example, if the following immediate mode line was entered, and the RETURN key pressed.

```
]PRINT "JIM SMITH"
```

the following would be displayed on the video screen:

```
JIM SMITH
```

In the program or indirect mode, the computer accepts program lines into memory, where they are stored for later execution. This stored program will be executed when the appropriate command (generally RUN) is entered.



```
]10 PRINT "JIM SMITH"
]20 PRINT "1220 EUCLID AVE"
]30 PRINT "CLEVELAND, OH 44122"
]40 END
]RUN
JIM SMITH
1220 EUCLID AVE
CLEVELAND, OH 44122
```

FIGURE 4-2. Program Mode Entry & Execution

Figure 4-2 contains an example of the entry of a program in the program mode and its execution.

Notice that in the program mode, that each BASIC prgram line must be preceded with a line number. Line numbers will be discussed in more detail later in this chapter.

COMMAND AND STATEMENT STRUCTURE

In SmartBASIC, instructions being relayed to the interpreter are known as **commands** in the immediate mode, and **statements** in the program mode. In practice, the difference between a command and a statement is primarily one of semantics, as both generally use the same structure and keywords.

Both commands and statements begin with a BASIC **keyword** or **reserved word.** The keyword identifies the operation to be undertaken by the BASIC interpreter. For example, in the preceding section, the PRINT command was used to instruct the ADAM to display information on the screen.

In SmartBASIC, keywords may be entered in either uppercase or lowercase letters. In the examples in this book, we will display keywords as uppercase letters.

A BASIC command or statement generally includes one or more **arguments** or **parameters** following the keyword. In our example,

```
PRINT "JIM SMITH"
```

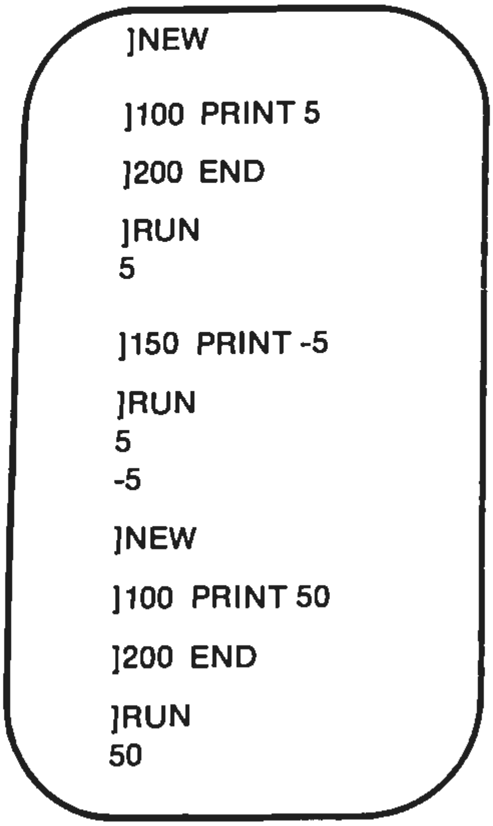
"JIM SMITH" is the PRINT statement parameter.

ENTERING A PROGRAM

In the preceding section, we touched upon the fundamentals of entering and running a BASIC program on the ADAM. In this section, we will expand upon that discussion using the example in Figure 4-3.

BASIC programs are entered as **program lines.** Any text preceded with a number (**line number**) and ended by pressing the RETURN key will be regarded as a program line.

The maximum number of characters that may be included in any one line is 128 including RETURN. If a line contains more than 128 characters, an error condition will result.



```
]NEW

]100 PRINT 5

]200 END

]RUN
5

]150 PRINT -5

]RUN
5
-5

]NEW

]100 PRINT 50

]200 END

]RUN
50
```

FIGURE 4-3. Entering and Running a Program

Note that in the first 5 lines of Figure 4-3, a program was entered in the command mode and run in the execute mode. After the answer, 5, had been displayed, the prompt appeared.

At this point, the original program is stored in memory, and can be run again if desired. Also, the program being held in memory can be added to or changed. That is what was done in line number 150 of Figure 4-3. An additional statement was inserted between statements 100 and 200 in the program being stored in memory. This revised program can be executed by again entering RUN.

The computer memory can only hold one program at a time. The NEW command is used to erase the program in memory so as to allow a new program to be entered. Note the use of NEW in Figure 4-3.

Note in our examples the following features common to the BASIC programs:

1. Each program line must begin with a line number. The computer executes program lines in order from lowest line number to highest line number.
2. The END statement signals the end of a program. When END is executed, the program run will stop.

It is recommended that consecutive line numbers (i.e. 1,2,3,4,5, etc.) not be used in a program. By using numbers which are a fixed multiple (i.e. 100, 200, 300, etc.), additional line numbers can be inserted between exisiting lines without renumbering the lines.

Line numbers need not be in any particular order. For example, the user could enter lines 100 and 200 and then enter line 150. The computer will automatically rearrange the lines according to their line numbers.

If the user enters two lines with the same line number, the computer will erase the first line and replace it with the second. This feature allows the user to replace an entire line by merely entering a new line with the same number.

A new line can be added to a BASIC program by merely entering a line number followed by the desired text and RETURN. When RETURN is pressed, the line will be saved as part of the BASIC program.

To delete a line in an existing program, merely enter the line number of the line to be deleted followed by RETURN. A group of lines can be deleted via the DEL command, and an entire program can be deleted with the NEW command. These will be discussed in detail in Chapter 12.

ERROR AND WARNING MESSAGES

When a statement with an incorrect format has been entered, an error message will be displayed. The error message describes the type of problem that occured.

If a problem develops while a program is being executed, an error message will also be displayed. An error that occurs during the execution of a program generates an error message that includes a description of the problem as well as the line number of the statement that caused the problem.

When an error occurs in a program, an error message will be displayed and its execution will stop. If a problem occurs in a program that is not serious enough to stop its execution, a warning message will be displayed. Warning messages describe the nature of the problem as well as the line number where the problem occurred.

LISTING A PROGRAM

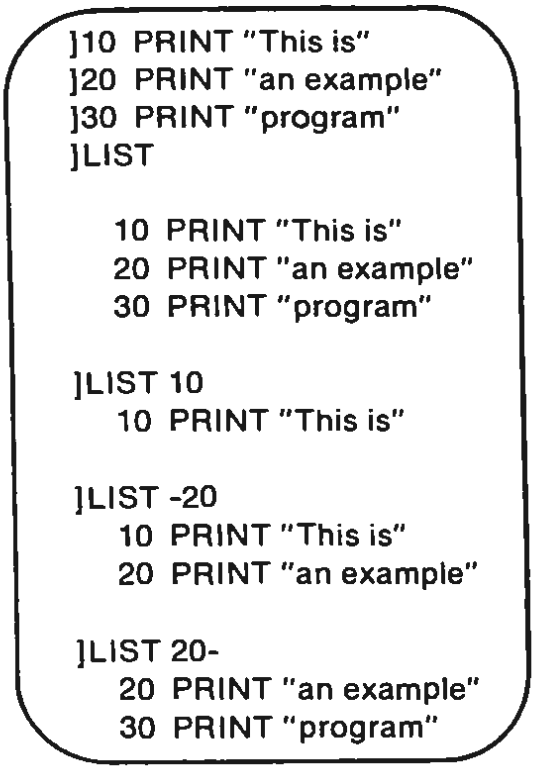
LIST is used to display the program stored in memory on the screen or printer. This display is often referred to as a **program listing.** An example of the use of LIST is given in Figure 4-4.

When the LIST command is executed, the program in the computer's memory will be displayed on the screen. Each program line will be displayed in increasing order of line numbers, regardless of the order of in which the program's lines were entered.

If a program occupies more than 24 display lines, the first lines of the program will be moved off the top of the display in order to accommodate the last lines. This process is called **scrolling.**

When a lenghty program is listed on the display, the information will only be displayed briefly. As a result, it is often necessary to halt the listing of a program. If this is the case, simply hold down the CONTROL Key and type the letter S. To continue the listing, simply repeat the CONTROL-S combination.

LIST can be used with optional parameters to display only a portion of the program. For example, LIST can be used with a single line number parameter to list only that line. This is shown in Figure 4-4.



```
]10 PRINT "This is"
]20 PRINT "an example"
]30 PRINT "program"
]LIST

   10 PRINT "This is"
   20 PRINT "an example"
   30 PRINT "program"

]LIST 10
   10 PRINT "This is"

]LIST -20
   10 PRINT "This is"
   20 PRINT "an example"

]LIST 20-
   20 PRINT "an example"
   30 PRINT "program"
```

FIGURE 4-4. Listing a Program

LIST can also be used with a range of line numbers. For example, the command LIST 10-30 would list all line numbers with values in the range 10 to 30.

LIST can be used to display all line numbers from the beginning of the program to a specified line by prefixing that line number with a hyphen and using it as the LIST parameter. For example, the following command:

```
LIST -150
```

would cause all program lines to be listed up to and including line 150.

If the line number parameter is followed by a hyphen, all program lines after and including the specified line will be listed. This is demonstrated in the final example in Figure 4-4.

EDITING A PROGRAM

If a program line is entered incorrectly it can be changed in one of two ways. The first method is to simply re-enter the program line. This is accomplished by retyping the line number, followed by one or more appropriate statements.

The second method uses the ADAM's full-screen edit feature to alter a program line. This feature allows the cursor to be moved to any location on the screen. Once the cursor has been located to an incorrect statement, the correct character or characters can be typed in place of the error.

The cursor can be moved by the 5 keys on the lower right portion of the keyboard. These keys are labeled ↑, ↓, ←, → and HOME. The "arrow" keys move the cursor in the direction of the arrow. The HOME key moves the cursor to the upper left corner of the display. The use of the full-screen editor is best explained by a simple example.

Begin by entering the following program:

```
]10 FOR T = 1 TO 100
]20 PRINT T
]30 NEXT T
```

When the LIST command is issued, the program will appear on the display as follows:

```
]LIST

  10 FOR t = 1 TO 100
  20 PRINT t
  30 NEXT t

]_
```

Suppose line number 10 is incorrect, and was intended to appear as follows:

```
10 FOR t =1 TO 200
```

This correction can be made by using the ↑ key to move the cursor to the first line of the program listing. Proceed by using the → key to move the cursor 18 spaces to the right. The cursor should now be directly under the 1 in the value 100, as follows.

```
10 FOR t = 1 TO 100
                ‾
```

Correct the error by typing the number 2 in the place of the 1. Continue by using the → key to move the cursor past the end of the program line, as follows.

```
10 FOR t = 1 TO 200 _
```

When a RETURN key is pressed, the corrected line will be added to the program in place of the incorrect statement. The RETURN key causes the cursor to move down 2 lines. When this occurs, the program line immediately after the line being edited will be cleared from the display. The screen will now appear as follows.

```
]LIST
   10 FOR t = 1 TO 200

]_ 30 NEXT t

]
```

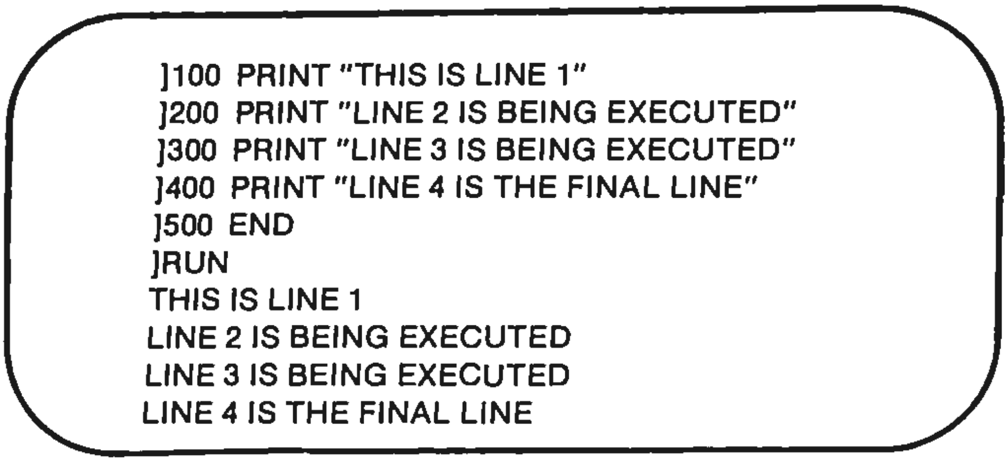
Even though line 20 no longer appears on the screen, it still exists in the program.

RUNNING A PROGRAM

Once a program is present in memory, the operator can run it. As mentioned previously, a program can be entered into memory via the keyboard or it can be loaded into memory from a Digital Data Pack. The procedure for loading program will be discussed later in this chapter.

The RUN command is used to begin program execution. RUN can be used with or without an optional line number or file specification as its parameter. Because RUN is generally executed without an optional parameter, we will limit our discussion to RUN in this section to its execution without parameters. The usage of RUN with parameters is discussed in Chapter 12.

When the RUN command has been entered and the Enter key pressed, program statements entered in the indirect mode (with line numbers) will be executed in order, beginning with the lowest line. An example of the usage of RUN is shown in Figure 4-4.



```
]100 PRINT "THIS IS LINE 1"
]200 PRINT "LINE 2 IS BEING EXECUTED"
]300 PRINT "LINE 3 IS BEING EXECUTED"
]400 PRINT "LINE 4 IS THE FINAL LINE"
]500 END
]RUN
THIS IS LINE 1
LINE 2 IS BEING EXECUTED
LINE 3 IS BEING EXECUTED
LINE 4 IS THE FINAL LINE
```

FIGURE 4-5. RUN Command

The execution of a program can be stopped at any time by holding down the CONTROL key and typing the letter C.

SAVING A PROGRAM

As you may recall from our discussion of program entry, only one BASIC program can be stored in memory at any one time. When the ADAM's power is turned off, its memory contents will be erased. Any program stored there will be lost unless it is first stored on a permament medium such as a Digital Data Pack.

Before a program can be saved, it must first be assigned a name consisting of one to ten characters. This name is known as a **filename.** Once a program has been assigned a filename, it can be saved with the SAVE command.

For example, the program in memory can be saved on the Digital Data Pack with the following command:

```
SAVE PROGRAM
```

When SAVE is executed, the program remains in memory where it can be added to, edited, or run if desired.

The Digital Data Pack is an effective means of retaining programs and data while the computer is turned off. The details of the procedures used to manipulate programs and data will be presented in Chapter 10.

LOADING A PROGRAM

Once a program has been saved on a Digital Data Pack, it can again be loaded back into memory using the LOAD command. An example of a LOAD command is given below:

```
LOAD PROGRAM
```

Before a program is loaded, any existing program in memory will be erased. Once the program has been loaded into memory, it can be edited or run as any other program would be.

If the file specified with LOAD is not present on the indicated diskette, the following error message will be displayed:

```
File not found
```

MULTIPLE STATEMENTS

In our examples thus far, we have only included one BASIC statement in each program line. In SmartBASIC, multiple statements may be included in a single program line as long as each of the statements are separated with a colon.

The following program uses multiple statements in line 10.

```
]10 PRINT "JOHN":PRINT "NELSON"
]20 PRINT "ATLANTA"
]30 PRINT "GEORGIA"
]40 END
]RUN
JOHN
NELSON
ATLANTA
GEORGIA
```

5

Data Types, Variables, & Operators

Introduction

In Chapter 4 we gained an overview of SmartBASIC and learned a few of SmartBASIC's fundamental operating details. In Chapter 5, we will begin learning the basic concepts necessary to master SmartBASIC. In this chapter, we will discuss the various types of **data** used in SmartBASIC as well as the various **operations** that can be performed on that data.

Data Types

The data processed in SmartBASIC can be classified under two special headings: string and numeric. String and numeric data are stored differently in memory by the ADAM. Also, the various **operators** in SmartBASIC affect string and numeric data in different manners.

STRINGS

A **string** can be defined as one or more **ASCII** characters. The various ASCII characters are listed in Appendix 3 and consist of the digits (0-9), letters of the alphabet, and a number of special symbols.

SmartBASIC also allows a string of zero characters. This is also known as the empty or null string and is used much as a zero is in mathematics.

As you may already have noted from our examples in Chapter 4, when a string is used in a SmartBASIC statement, it must be enclosed within quotation marks. The quotation marks serve to identify the beginning and ending points of the string. They are not a part of the string.

A string enclosed within quotation marks is known as a **string constant.** A **constant** is an actual value used by BASIC during execution. The following are examples of string constants.

```
"JOHN SMITH"
"12197"
"E97432"
"BOSTON, MA 01270"
"213-729-4234"
```

Notice that numbers can be used within a string constant. Remember, however, that the numbers within a string constant are string rather than numeric data.

One final point that should be kept in mind regarding string constants is that they cannot contain quotation marks. For example, the following string constant:

```
"John said, "Goodbye." as he walked away."
```

would be illegal. Since quotation marks are used to denote the beginning and ending points of a string constant, their inclusion within the string itself would cause difficulties and therefore is not allowed.

In Chapter 9 we will discuss how the CHR$ function can be used to place the ASCII code for quotation marks within a string constant.

NUMERIC DATA

Numeric data can be defined as information denoted with numbers. Numeric data is stored and operated on in a different manner than is string data.

Numeric constants consist of positive and negative numbers. Numeric constants cannot include commas. For example, 10,000 would be an illegal numeric constant.

SmartBASIC further classifies numeric constants as **integers, fixed-point** numbers and **floating-point** numbers.

Integers can be defined as the whole numbers in the range between -32767 and 32767 inclusive. Integer numbers do not have a decimal portion. Fixed-point numbers can be defined as the set of positive and negative real numbers. Fixed-point numbers contain a decimal portion. Floating-point numbers are represented in scientific notation. A number in scientific notation takes the following format:

$$\pm\, x \,\mathrm{E} \pm yy$$

Where;

± is an optional plus or minus sign.

x can either be an integer or fixed point number. This position of the number is known as the coefficient or mantissa.

E stands for exponent.

yy is a two digit exponent. The exponent gives the number of places that the decimal point must be moved to give its true location. The decimal point is moved to the right with positive exponents. The decimal point is moved to the left with negative exponents.

The following are examples of floating-point numbers and their equivalent notation in fixed-point.

Floating-Point	Fixed-Point
3.87E+05	387000
4.064E-04	.0004064
-1E+06	-1000000
7.87642E+03	7876.42

SmartBASIC can only handle floating point numbers in the range between 1.70141183E+38 and -1.70141183E+38. Any decimal numbers in the range between -2.93873587E-39 and 2.93873587E-39 will be converted to zero.

Floating-point notation is used as a more efficient means for the computer to manipulate exceedingly large or exceedingly small values. As a result, some values that are entered in fixed-point notation may automatically be converted to floating-point notation by the computer.

Numbers represented in floating-point or fixed-point notation contain a maximum of 9 digits of accuracy. Any additional digits in a number will be truncated.

Integers differ from floating-point and fixed-point values in the fact that integers cannot contain digits to the right of the decimal point. This condition allows integers to be stored in a smaller area of the computer's memory. Also, integers can be handled more quickly than other types of values.

The following are examples of integers, floating-point and fixed-point numeric values.

Integers	**Floating-Point**	**Fixed-Point**
-7978	-387E+04	47988
32600	4.015E+07	37.0
37	6.870E-27	45.874
192	1E+06	3.1415927
-687	1.414E+00	-238.5

Note that 47988 cannot be considered an integer since it lies outside the allowable range of values (-32767 to 32767). Also, note that 37.0 is not an integer because it contains a digit to the right of the decimal point.

Variables — An Overview

In the preceeding section, we discussed SmartBASIC's different types of data — string and numeric. In the remainder of this book, we will refer to a string constant as a string and a numeric constant as a number.

So far, we have only discussed representing data as a constant. The value of a string or numeric constant such as "JIM HILL" or 27.92 always remains the same.

Data can also be represented by using a **variable**. A variable can be defined as an area of memory that is represented with a name. That name is known as the **variable name**. The information stored in the memory area defined by a variable name can vary (hence the name variable) as SmartBASIC commands or statements are executed. The data currently stored in the memory area defined by a variable is known as the variable's **value.**

VARIABLE NAMES

SmartBASIC allows variable names of up to 25 characters in length. A variable name must begin with a letter of the alphabet (A-Z) followed by additional letters or digits. Blank spaces are not allowed within a variable name. Letters entered in uppercase are automatically converted to lowercase by the computer. The following are examples of valid SmartBASIC variable names

```
abc        x9
name       address
```

Although 25 characters can be used in a variable name, the computer can only recognize the first two characters. As a result, the computer will not be able to differentiate between two variables such as AB1 and AB2.

A variable name may not duplicate a SmartBASIC reserved word (see Appendix 1). However, a variable name may incorporate a reserved word as part of its name.* Therefore, although the following would be invalid variable names:

```
NEW          AND          PRINT
```

the following variable names would be valid:

```
NEWPHONE     ANDY     PRINTNAME
```

Variables, like constants, can either be numeric or string. Nu variables can be integer or decimal values.

A variable type can be declared by using a type identification character. The type identification characters are as follows:

% = integer
$ = string

For example, the following variable names,

```
ANCIENT$          JACK%
```

*The exception to this rule is FN. A variable name may not begin with FN.

would be declared as string, and integer respectively. If a variable type character is not specified, the variable is assumed to be a decimal value.

INITIAL VARIABLE VALUES

Numeric variables are initially assumed to have a value of zero. String variables are initially assumed to be null. Values may be assigned to a variable as the result of a calculation or as the result of an assignment statement (discussed later).

SmartBASIC does not allow a string value to be assigned to a numeric variable or vice versa. However, a decimal value can be assigned to an integer value. In this case, the decimal portion of the value will be neglected and only the integer part will be assigned to the integer variable, as shown in the following example.

```
]10 X = 3.1415927
]20 X% = X
]30 PRINT X%
]40 END
]RUN
3
```

In a similar manner, integers can be assigned to decimal variables. The following example demonstrates the result.

```
]10 X% = 5280
]20 X = X%
]30 PRINT X
]40 END
]RUN
5280
```

ASSIGNMENT STATEMENTS (LET)

The LET statement is used to assign a value to a variable. LET statements are also known as **assignment** statements. LET is used with the following configuration:

LET *variable = expression**

*In our configuration examples, BASIC reserved words will be depicted in upper-case, regular-face type. Parameters to be entered by the programmer will be depicted in lower-case italics.

Whenever a LET statement is used in a program, the value of the variable on the left side of the equation is replaced with the value appearing on the right.

The reserved word, LET need not actually be included in a LET statement. Both of the following statements have the same meaning:

```
100 LET A = 5
200 A = 5
```

The value assigned to a variable can either be a constant, a variable, or the result of an operation. In the following example, A$ is assigned the string constant "JOHN", B is assigned the numeric constant 27.9, C is assigned the value of B, and D is assigned the value of B multiplied by 2.

```
]10 A$ = "JOHN"
]20 B = 27.9
]30 C = B
]40 D = B * 2
]50 PRINT A$
]60 PRINT B
]70 PRINT C
]80 PRINT D
]RUN
JOHN
27.9
27.9
55.8
```

CLEAR Statement

A CLEAR statement has the opposite effect of an assignment statement. CLEAR statements cause the values of all numeric variables to be set to zero, and all string variables to be set to null (no characters).

Expressions & Operators

The values of variables and constants are combined to form a new value through the use of **expressions.** The following are examples of expressions.

```
4 + 7
A$ + B$
3 * 1
14 < 21
X AND Y
```

SmartBASIC includes several types of expressions including **arithmetic, relational, and Boolean.** In our previous examples, the first three examples were arithmetic expressions, while the fourth and fifth were examples of relational and Boolean expressions respectively. Each of these types of expressions will be discussed in detail in the following sections.

The sign or word describing the operation to be undertaken is known as the **operator.** An operator is a symbol or word which represents an action which is to be undertaken on one or more values specified with the operator. These values are known as operands.

The operators in our previous examples were as follows:

```
+
+
*
<
AND
```

ARITHMETIC OPERATORS

Arithmetic operators are used to perform mathematical operations on numeric or string variables. The arithmetic operators are listed in Table 5-1.

Table 5-1 Order of Evaluation

Symbol	Operation	Example
-	Negation	-A
^	Exponentiation	A∧B
*	Multiplication	A * B
/	Division	A/B
+	Addition	A + B
—	Subtraction	A — B

The first arithmetic operation specified in Table 5-1 is exponentiation. Exponentiation is the process of raising a number to a specified power. For example, in the following,

$$A^5$$

the expression would be evaluated as:

A * A * A * A * A

In SmartBASIC, exponentiation is indicated with the caret symbol, ^.

Exponentiation can be used in an arithmetic expression as shown below:

7∧2

The preceding expression would evaluate to 49.

The second operation indicated in Table 5-1 is negation. When the -sign precedes a number, the symbol is used to change that number's sign. This usage is known as negation.

The symbols + and - are used for addition and subtraction respectively. The asterisk (*) is used to indicate multiplication, while the slash (/) is used to indicate division.

ORDER OF EVALUATION (ARITHMETIC EXPRESSIONS)

All of our preceding examples were **simple expressions**. A simple expression is one which contains just one operator and one or two operands. Simple expressions can be combined to form **compound** expressions. The following are examples of compound expressions:

```
(A + B) * 7 - 4
(A + B) * A * (C + D)
27 + 47 ^ A - B
```

With compound expressions, it is necessary that the computer knows which operations should be undertaken first. SmartBASIC follows a standard order of evaluation within compound expressions.

In this section, we will discuss the order of evaluation of compound arithmetic expressions. Later in this chapter, we will discuss the order of evaluation of relational and logical operators as well as the overall order of evaluation of arithmetic, relational, and logical operators as a group.

In an expression with more than one arithmetic operator, the operators with higher priority are evaluated first followed by those with lower priority. Evaluation is accomplished from left to right in the expression. The following is an example of the evaluation of the arithmetic operators in an expression.

```
A = 37.1 + 12.9 * 2.1 + 7 - 4 ^ 2
  = 37.1 + 12.9 * 2.1 + 7 - 16
  = 37.1 + 27.09 + 7 - 16
  = 55.19
```

Parentheses can be used to alter the order of evaluation in arithmetic expressions. Expressions appearing within parentheses have the highest priority in the order of evaluation. For example, the use of parentheses with our preceding example could change the value of the expression.

```
A = (37.1 + 12.9) * 2.1 + (7 - 4) ^ 2
  = 50.0 * 2.1 + 3 ^ 2
  = 50.0 * 2.1 + 9
  = 105.0 + 9
  = 114.0
```

MIXING VARIABLE TYPES IN ARITHMETIC EXPRESSIONS

Although certain variable types may be mixed in a SmartBASIC expression, it is preferable to use a single variable type throughout each expression. By doing so, execution time will be decreased, memory requirements will be reduced, and the probability for program errors will be reduced.

An example of mixing different numeric type in the same expression is given below:

A = B + 1

Both A and B are numeric variables, while 1 is an integer constant. The integer constant must be converted to a real number (as shown below), before the expression can be evaluated.

A = B + 1.0

When numeric variables are used in expressions, the variable on the left side of the equal sign is assigned the value of the expression on the right side. The value will be converted to the value specified by the numeric variable on the left side. For instance, in the following,

A% = 1.03 + 2.07

the value of the expression on the right hand side (3.1) will be converted to the integer 3 so that it agrees with the integer variable A%. If A% were replaced with A, no conversion would have been necessary.

RELATIONAL OPERATORS

Relational operators are used to make a comparison using two operands. The following relational operators are used in SmartBASIC.

<	→	less than
<=	→	less than or equal to
>	→	greater than
>=	→	great than or equal to
=	→	equal to
<>	→	not equal

A relational operation evaluates to either true or false. For example, if the constant 1.0 was compared to the constant 2.0 to see whether they were equal, the expression would evaluate to false. In SmartBASIC, a value of 1 represents a condition of true, while a value of 0 represents false.

The only values returned by a comparison in SmartBASIC are 1 (true) or 0 (false). These values can be used as any other integer would be used. The following results are generated by the following relational expressions:

$5 > 7 \longrightarrow 0$ (false)

$5 > 3 \longrightarrow 1$ (true)

$7 = 7 \longrightarrow 1$ (true)

Relational operations are evaluated after the addition and subtraction arithmetic operations. Relational operators are always performed from left to right in an expression.

Although different numeric data types may be compared in a relational expression, numeric data may not be compared to string data.

Relational operations using numeric operations are fairly straightforward. However, relational operations using string values may prove confusing to the first-time user. Strings are compared by taking the ASCII value for each character in the string one at a time and comparing the codes.

For example, consider the two string values "JOSEPH" and "JOAN". In a relational expression, the first characters of the strings will be compared first. Since both strings begin with "JO", the comparison will continue with the third character.

Since the ASCII code for "A" (65) is less than the ASCII code for "S" (83), "JOAN" is considered less than "JOSEPH".

If the end of a string is encountered during a string comparison, the string with the fewer number of characters will be considered less than the longer string. For example, "ABC" would be considered less than "ABCD". The relational operators can be used in this manner to indicate the relative location of strings in alphabetical order.

The following examples demonstrate the use of relational operators with string values. All of the following expressions are true.

```
"ABC" = "ABC"
"ABC" > "AAA"
"ALFRED" < "ALFREDO"
A$ < Z$ where A$ = "ALFRED" and Z$ = "ALFREDO"
```

Note that all string constants must be enclosed in quotation marks.

LOGICAL OPERATORS

Logical or Boolean operators are generally used in SmartBASIC to compare the outcomes of two relational operations. Logical operations themselves return a true or false value which will be used to determine program flow.

The logical operators are NOT (logical complement), AND (conjunction), and OR (disjunction). These return results as shown in Figure 5-1.

A logical operator evaluates an input of one or more operands with true or false values. The logical operator evaluates these true or false values and returns a value of true or false itself. An operand of a logical operator is evaluated as true if it has a non-zero value (Remember, relational operators return a value of 1 for a true value.). An operand of a logical operator is evaluated as false if it is equal to zero.

The result of a logical operation is also a number which, if non-zero is considered true, and false if it is zero.

The following are examples of the use of logical operators in combination with relational operators in decision-making:

```
IF X > 10 OR Y < 0 THEN 900
IF A > 0 AND B > 0 THEN 200
FLAG% = NOT FLAG%
```

In the first example, the result of the logical operation will be t. variable X has a value less than 0. Otherwise, it will be false. If the result of the logical operation is true, the program will branch to line 900. Otherwise, it will continue to the next statement.

In the second example, the result of the logical operation will be true only if the values of both variables A and B are greater than zero. If the result of the logical operation is true, program control will branch to line 200.

In the final example, the value of FLAG% is switched from true to false or vice versa.

Figure 5-1 contains tables that may prove helpful when evaluating program statements using logical operators in combination with relational operators.

NOT Operation

T	F	A Operand
F	T	NOT A

OR Operation

T	T	F	F	A Operand
T	F	T	F	B Operand
T	T	T	F	A OR B

AND Operation

T	T	F	F	A Operand
T	F	T	F	B Operand
T	F	F	F	A AND B

FIGURE 5-1. Logical Operators

The NOT, AND, and OR operators are best explained with simple analogy. Suppose that Steve and Sherry were shopping in the produce department of their grocery store. If they decided to collectively purchase an item if either of them individually wanted that item, they would be acting under the OR logical operator.

Now, suppose that Steve and Sherry decided that they would only purchase an item if they both wanted that item. They would then be acting under the AND logical operation.

Now, suppose that Sherry was angry with Steve. If Sherry decided not to purchase the items that Steve wanted, but purchased everything he didn't want instead, she would be acting under the NOT logical operation.

ORDER OF EVALUATION — OVERVIEW

Earlier in this chapter, we outlined the order of evaluation within an expression with respect to arithmetic operators. Now that we have introduced the concepts of relational and logical operations, we can revise our order of evaluation to that of Table 5-2.

Table 5-2. Order of Evaluation

	Operator	Description	Priority
Parentheses	( )	Used to alter order of evaluation.	1
Unary Operators	— NOT	Unary Minus Logical Complement	2
Arithmetic Operators	^	Exponentiation	3
	• /	Multiplication Division	4
	+ -	Addition Subtraction	5
Relational Opertors	= <> < > <= >=	Equal To Not Equal To Less Than Greater Than Less Than or Equal To Greater Than or Equal To	6
Logical Operators	AND	Conjuction	7
	OR	Disjunction	8

The unary operators will always be performed before any other operators. Exponentiation will be performed next, followed by multiplication and division. Addition and subtraction will always be the last arithmetic operations to be performed. Relational operations will be performed after the arithmetic operations, but before AND and OR.

In the absence of parentheses, operators with the same priority will be performed from left to right in an expression.

6

Inputting and Outputting Data

Introduction

In Chapters 4 and 5, we briefly described the usage of the PRINT statement to output data. In this chapter, we will discuss the usage of PRINT for outputting data to the screen or printer in depth.

After we have discussed the methods used in SmartBASIC to output data, we will discuss the statements used to input data into variables. These include INPUT and GET.

PRINT

To this point, we have only used the PRINT statement to output a single constant or variable value to the screen. The PRINT statement can also be used to output more than one item to the screen. When PRINT is used in this manner, the spacing between the items to be printed can be controlled by separating them with a comma or semicolon. For example, compare the results of the following PRINT statements.

```
]PRINT "BILL" "STEVE" "LARRY"
BILLSTEVELARRY
]PRINT "BILL","STEVE"
BILL          STEVE
]PRINT "BILL";"STEVE";"LARRY"
BILLSTEVELARRY
```

Notice that in our first example, no delimiter was used to separate the three string constants. These were output as one continuous string.

In the second example, the comma was used to delimit the string constants. When a comma appears in a PRINT statement, the computer is instructed to begin printing the next parameter in the PRINT statement at the beginning of the next print zone. SmartBASIC divides the spacing on a line into two print zones. The first print zone extends from the left-most edge of the display to the middle of the screen. The second print zone extends from the center of the line to the right hand edge of the screen.

Commas are very useful when data is to be output in tabular form. This is illustrated in the following example program.

```
]10 PRINT "Name","ID No."
]20 PRINT "Jack Williams",3749
]30 PRINT "Ann Timmons",3622
]40 PRINT "Jay Randolph",2511
]50 END
]RUN
Name                ID No.
Jack Williams       3749
Ann Timmons          3622
Jay Randolph        2511
```

In the third example at the top of this page, the semicolon was used as the delimiter. The semicolon causes each string data item in the PRINT statement to be output immediately adjacent to the preceding item.

When semicolons are used to separate data items in a PRINT statement, the output will be displayed without the insertion of any additional spaces between data items. As a result, spaces must be inserted in PRINT statements between any data items that need to be separated. The most common technique used to insert spaces is to include a space (enclosed in quotation marks) in a PRINT statement. The following example program demonstrates this technique.

```
]10 A$ = "COLECO"
]20 B$ = "ADAM"
]30 PRINT A$;B$
]40 PRINT A$;"  ";B$
]50 END
]RUN
COLECOADAM
COLECO ADAM
```

Another technique is also commonly used to insert spaces in data being output. If a space is assigned to a string variable, a space will be output each time the variable is included in a PRINT statement.

The following example program demonstrates this concept.

```
]10 A$ = "  "
]20 A = 5
]30 B = 8
]40 PRINT A;A$;B;A$;A/B
]50 END
]RUN
5 8 .625
```

Notice that the variable A$ is assigned a single blank character. A space is inserted in the output whenever A$ appears in a PRINT statement.

Note also that when an arithmetic expression is specified in a PRINT statement, that expression's result will be output.

Generally, when a PRINT statement has been executed, the cursor or print head will advance to the leftmost position on the next output line. This is known as a carriage return/line feed, which can be abbreviated as CR/LF.

A CR/LF can be suppressed by ending a PRINT statement with either a comma or a semicolon. When a semicolon is used to end a PRINT statement, the output from the next PRINT statement will be positioned immediately after the data output by its predecessor. This is illustrated in the following example.

```
]10 PRINT "Data1";
]20 PRINT "Data2";
]30 PRINT "Data3";
]40 END
]RUN
Data1Data2Data3
```

When a PRINT statement ends with a comma, subsequent data will be output at the next print zone on the same display line. This is shown in the following example.

```
]10 PRINT "Data1",
]20 PRINT "Data2",
]30 END
]RUN
Data1          Data2
```

There are several features that can be used in the outputting of data. The first of these features is the inverse mode. When an INVERSE statement is executed, the subsequent output appears as dark characters against a light background. The inverse mode will be in effect until a NORMAL statement is executed. The following example program demonstrates the inverse mode of output.

```
]10 A$ = "OUTPUT"
]20 PRINT A$
]30 INVERSE
]40 PRINT A$
]50 NORMAL
]60 PRINT A$
]70 END
```

Horizontal Formatting

SmartBASIC includes several functions that allow the programmer to control the horizontal format of output. These include TAB and SPC.

TAB

BASIC allows an item to be printed in any position on the screen or printer with the TAB command. The print position can range from 1 to 255.

```
]10 PRINT TAB(10)"10";TAB(200)"200"
]RUN
          10



             200

]
```

Notice that the example caused output to be positioned at columns 10, and 14. Earlier, we mentioned that print positions can range from 1 to 255. These 255 print positions are the result of the fact that a logical line in BASIC can consist of up to 255 characters. If the argument of a TAB function exceeds the length of the display line, the output will occur on a subsequent line. This explains why the output of the preceding example program requires eight lines of the display.

SPC

SPC causes the number of spaces specified as its argument to be sent to the display or printer.

```
]10 PRINT "JOHN" SPC(10) "FLETCHER"
]RUN
JOHN          FLETCHER
```

In our preceding example, SPC(10) causes the cursor to move 10 positions to the right once JOHN has been output.

VERTICAL AND HORIZONTAL TABS

The output of a program can be tabulated by either rows or columns on the display. This can be accomplished with appropriate VTAB and HTAB statements. An HTAB statement causes the cursor to move to a specified column on the display. Similarly, a VTAB statement causes the cursor to proceed to a specified row. The specified row or column number must be included immediately after the VTAB or HTAB keyword. The following example program uses the tabulation features to output data in the four corners of the display.

```
]10 HOME
]20 PRINT "upper left";
]30 HTAB 21
]40 PRINT "upper right"
]50 VTAB 20
]60 PRINT "bottom left";
]70 HTAB 20
]80 PRINT "bottom right"
]90 END
```

The HOME statement at line 10 causes the screen to be cleared. As a result, the first output will occur in the upper left corner of the display. The HTAB statement at line 30 causes the cursor to move to column 21 in the first row. The second output will then occur in the upper right of the display. The VTAB statement at line 50 causes the cursor to proceed to row number 20 on the display. The final HTAB statement causes the cursor to move near the lower right corner of the display, where the final output will occur.

Outputting Data to the Printer

Normally, a PRINT statement causes data to appear on the display. However, data can be output to the printer if appropriate commands are used.

The first method that can be used to output data is the use of the CONTROL key along with the letter P. Whenever the P key is typed while the CONTROL key is being held down, the data that appears on the display will be output to the printer. Any characters that appear on the display will be printed exactly as they appear.

The second method that can be used to output data to the printer is the PR#1 command. When this command is executed in the immediate mode, or as a program statement, any subsequent PRINT or LIST statements will cause data to be sent to the printer instead of the display. Similarly, the PR#0 command can be used to deactivate the printer.

In order to become acquainted with the use of the printer, execute the commands described below. Begin by executing the NEW and HOME commands. Proceed by entering the following program.

```
]10 PR#1
]20 FOR T = 1 TO 10
]30 PRINT T
]40 NEXT T
]50 END
```

Be sure that the printer is ready for operation and press the P key while holding down the CONTROL key. The program will immediately be output to the printer exactly as it appears on the display. Continue by entering the RUN command.

```
]RUN
```

The program will be executed and the output will be sent concurrently to the display and the printer. The PR#1 command was used in the program to activate the printer. The command will remain in effect until the PR#0 command has been issued. Entering the LIST command while the printer is activated will cause the program listing to be output once again. The PR#0 command can be used to deactivate the printer at any time.

Inputting Data

Data can be assigned to a variable while a program is being executed. This can be accomplished with the INPUT or GET statements.

INPUT

When an INPUT statement is executed, the computer will display a question mark and wait for the operator to enter a response. That entry will be assigned to the variable indicated. The entry must be ended by pressing the RETURN key. Program execution will then resume.

The values of several variables can be input with a single INPUT statement. These variables may either be numeric or string as shown in the following example:

```
100 INPUT A$,B$,C
```

When the preceding INPUT statement is executed, the INPUT prompt (?) will be displayed. The operator should then input the data items for variables A$, B$, and C. Each input should be separated by a comma. The RETURN key should be pressed after all input entries have been made. An example of a valid entry for the preceding INPUT statement is given below:

```
JOHN,SMITH,281
```

These entries will be assigned to the variables as follows:

```
A$ = "JOHN"
B$ = "SMITH"
 C = 281
```

If an incorrect number of entries were made or if a string constant were input for a numeric variable or vice versa, the following message would be displayed,

```
? Reenter
?
```

and the computer would wait for a valid entry.

It is good programming practice to include a prompt message with the INPUT statement to let the operator know what data the computer is expecting. For example, the following INPUT statement:

```
100 INPUT "ENTER COMPANY NAME,NUMBER";A$,B
```

would result in the following prompt being displayed:

```
ENTER COMPANY NAME,NUMBER
```

A typical response would be as follows:

```
ACME MFG,27
```

Notice that we did not need to enclose our string entries within quotation marks. When a string is being entered in response to an INPUT prompt, the quotation marks can be excluded unless the entry includes commas or begins with one or more blank spaces.

GET STATEMENTS

A GET statement is an alternate means of inputting data from the keyboard. A GET statement can be used to accept a single character, and assign that character to a string variable. When a GET statement is executed, the execution of the program will pause until a key on the keyboard has been pressed. Unlike a response to an INPUT statement, the response to a GET statement does not appear on the display and does not need to be followed by pressing RETURN. A typical GET statement would appear as follows.

```
GET X$
```

A GET statement can also be used to assign a digit to a numeric variable. When a GET statement is used in this fashion, it cannot accept any character other than the digits 0 through 9. The following two example programs demonstrate typical applications of a GET statement.

```
]10 PRINT "This is Part I"
]20 PRINT "Press any key to continue"
]30 GET DUMMY$
]40 PRINT "This is Part II"
]50 END

]10 PRINT "Make a Selection (0-9)"
]20 GET X
]30 PRINT "Selection ";X;" was chosen"
]40 END
```

The GET statement in the first example program can accept any character as a response. The GET statement in the second example program can accept only a single numeric digit. If a non-numeric character is entered in response to the second example statement, the following error message will be displayed.

```
? Syntax Error
```

7

Conditional, Branching, and Looping Statements

Introduction

Thus far in our discussion of SmartBASIC, our statements have been executed in sequential order. Several BASIC statements are available which can be used to alter program control. These include:

```
GOTO            IF, THEN
GOSUB           ONERR GOTO
ON, GOTO        FOR, NEXT
ON, GOSUB
```

These statements will be discussed in the following sections.

CONDITIONAL BRANCHES

One of the most important features of a computer is its ability to make a decision. BASIC uses the IF, THEN statement to take advantage of the computer's decision making ability. The IF, THEN statement takes the following form:

IF *expression* THEN *statement*

The IF statement sets up a question of a condition. If the answer to that question is true, the *statement* following THEN will be executed. If the answer is false, the subsequent program statement will be executed.

In the following example, if X is equal to 1, then Y will be set to 1.

```
100 IF X = 1 THEN Y = 1
```

The ON statement can also be used with GOTO or GOSUB to set up a condition which will branch program control. ON, GOTO and ON, GOSUB will be discussed later in this chapter.

BRANCHING STATEMENTS

Branching statements change the execution pattern of programs from their usual line by line execution in order of ascending line number. A branching statement allows program control to be altered to any line number desired. The most commonly used branching statements in BASIC are GOTO and GOSUB.

GOTO takes the following format:

GOTO *line number*

For example, the following program statement,

```
500 GOTO 999
 •
 •
 •
999 END
```

to branch program control at line 500 to line 999.

SUBROUTINES AND GOSUB

Many times you will find that the same set of program instructions are used more than once in a program. Re-entering these instructions throughout the program can be very time consuming. By using **subroutines,** these additional entries will be unnecessary.

A subroutine can be defined as a program which appears within another larger program. The subroutine may be executed as many times as desired.

The execution of subroutines is controlled by the GOSUB and RETURN statements. The format for the GOSUB statement is as follows:

GOSUB *line number*

The computer will begin execution of the subroutine beginning at the *line number* indicated. Statements will continue to be executed in order, until a RETURN statement is encountered. Upon execution of the RETURN statement, the computer will branch out of the subroutine back to the first line following the original GOSUB statement. This is illustrated in the following example.

```
] 10  GOSUB 100
] 20  GOSUB 200
] 30  END
]100  PRINT "subroutine #1"
]110  RETURN
]200  PRINT "subroutine #2"
]210  RETURN
]RUN
subroutine #1
subroutine #2
```

Subroutines can help the programmer organize his program more efficiently. Subroutines also can make writing a program easier. By dividing a lengthy program into a number of smaller subroutines, the complexity of the program will be reduced. Individual subroutines are smaller and, therefore, more easily written. Subroutines are also more easily debugged than a longer program.

CONDITIONAL STATEMENTS WITH BRANCHING

Branching statements are often used in conjuction with conditional statements. In such a situation, the normal execution of the program is altered depending upon the outcome of the condition set up in an IF or an ON statement. This is shown in the following example.

```
]100 INPUT "Enter the amount."; a
]200 IF a = 0 THEN GOTO 500
]300 PRINT a
]400 GOTO 100
]500 INPUT "Are you finished?";a$
]600 IF a$ <> "y" THEN 100
]700 END
```

In our preceding example, if the value input for A has a zero value, then the program will branch to line 500 where the operator will be asked whether he has finished entering data. In line 600, the program will set up a condition where if the input was anything other than the letter "y", the program will branch to line 100. If the entry was equal to y, the program will end at line 700.

Note in line 600 that a GOTO statement is not used to precede the line number being branched to. When a line number is indicated following a THEN statement, the computer assumes the presence of GOTO.

The ON, GOTO and ON, GOSUB statements are also combinations of a conditional statement and a branching statement. The use of the ON, GOTO statement is illustrated in the following program.

```
]10 INPUT A
]20 ON A GOTO 40,50
]30 GOTO 99
]40 PRINT "A = 1":GOTO 99
]50 PRINT "A = 2"
]99 END
```

If the variable or expression following ON evaluates to 1, program control branches to the first line number specified after GOTO; if 2, to the second, etc.

If the variable or expression evaluates to a number greater than the number of line numbers following GOTO, program control will branch to the statement immediately following the ON, GOTO statement. This is also the case if the variable or expression following ON evaluates to zero. Negative values for the control expression are not allowed.

The ON, GOSUB statement is very similar in nature to the ON, GOTO statement. The following statement is an example of an ON, GOSUB statement.

```
100 ON X GOSUB 1000,2000,3000
```

If the value of X is 1, the subroutine at line 1000 will be executed. If X is 2, the subroutine at line 2000 will be executed. If X is 3, the subroutine at line 3000 will be executed. If X evaluates to 0 or to a number greater than 3, the statement immediately following the ON, GOSUB will be executed.

If ON,GOSUB causes a branch to a subroutine, program control will revert to the line immediately following the ON,GOSUB statement, once the subroutine has been executed.

LOOPING STATEMENTS

Suppose that you needed to compute the square of the integers from 1 to 20. One way of doing this is by calculating the square for each individual integer as shown below.

```
]100 A = 1 ∧ 2
]200 PRINT A
]300 B = 2 ∧ 2
]400 PRINT B
]500 C = 3 ∧ 2
]600 PRINT C
      .
      .
      .
```

This method is very cumbersome. The problem could be solved much more efficiently through the use of a FOR,NEXT loop as shown below.

```
]100 FOR A = 1 TO 20
]200 X = A ∧ 2
]300 PRINT X
]400 NEXT A
]500 END
```

The sequence of statements from line 100 to 400 is known as a **loop.** When the computer encounters the FOR statement in line 100, the variable A is set to 1. X is then calculated and displayed in lines 200 and 300.

The NEXT statement in line 400 will request the next value for A. Execution returns to line 100 where the value of A is incremented by 1 (to 2) and then compared to the value appearing after TO. Since the value of

A is less than that value, the loop will be executed again with the value of A set at 2.

The loop will continue to be executed until A attains a value greater than 20. When this occurs, the statement following the NEXT statement will be executed.

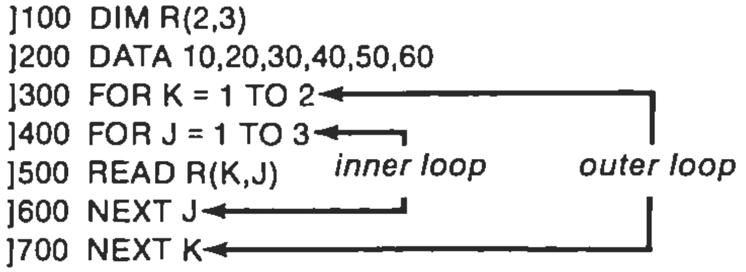
In our preceding example, A is known as an **index variable.** If the optional keyword STEP is not included with the FOR statement, the index variable will be incremented by 1 every time the NEXT statement is executed.

STEP can be included at the end of a FOR statement to change the value by which the index variable is incremented. The integer appearing after STEP is the new increment. For example, if our preceding example were changed as follows,

```
]100 FOR A = 1 TO 20 STEP 2
]200 X = A ∧ 2
]300 PRINT X
]400 NEXT A
]500 END
```

the index variable, A, would be incremented by 2 every time the NEXT statement was executed.

One loop can be placed inside another loop. The innermost loop is known as a **nested** loop. The following program contains a nested loop.



```
]100 DIM R(2,3)
]200 DATA 10,20,30,40,50,60
]300 FOR K = 1 TO 2
]400 FOR J = 1 TO 3
]500 READ R(K,J)
]600 NEXT J
]700 NEXT K
```

Our preceding example is used to read data into the numeric array R. Arrays as well as the READ and DATA statements will be discussed in Chapter 8.

One error that you should take care to avoid when using nested loops is to end an outer loop before an inner loop is ended. Also, be certain that every NEXT statement has a matching FOR statement. If the BASIC interpreter cannot match every NEXT statement with a preceding FOR statement, an error will result.

Error Handling

In some situations, it is easier to correct problems as they occur in a program, rather than to avoid them. This technique is called **error handling.** SmartBASIC allows the use of an ONERR GOTO statement to specify a line number where the program should proceed if an error occurs. This feature allows a portion of the program to be set aside as an error handling routine.

Error handling routines are commonly used to correct small problems that occur infrequently in a program. When appropriate corrections have been performed, a RESUME statement can be used to branch the program back to the location where the error occurred. The following program demonstrates the technique used to branch a program in the event of an error.

```
]10  ONERR GOTO 100
]20  INPUT "INPUT x:";x
]30  y = x  ^  .5
]40  PRINT "The square root of";x; "is" ;y
]50  END
]100  y = -x  ^  .5
]110  PRINT "The square root of ";x;" " is " ;y;"i"
]120  END
```

The preceding example program contains an ONERR GOTO statement at line 10. This statement indicates that the program control will branch to line 100 in the event of an error. An ONERR statement must be executed in a program before an error actually occurs.

The program calculates the square root of a value input for the variables x. However, BASIC does not allow the square root of negative numbers. These values can only be defined in the context of complex numbers, where the symbol "i" is used to represent the square root of -1. As a result, the square root of -4 could be represented by the value 2i since the following expression is true.

$$\sqrt{-4} = \sqrt{4}\,\sqrt{-1} = \sqrt{4}\;\; i = 2i$$

It is not necessary to understand the use of "complex" numbers to understand the example. The main concept of the program is that the statement at line 30 would normally have caused an error if a negative value had been input for the variable x. However, in this case, the ONERR statement causes the program to branch to line 100 whenever an error occurs.

Lines 100 and 110 perform an alternate set of operations whenever a negative value is input for x. Some typical applications of the sample program would appear as follows.

```
]RUN
Input x: 4←——user's response
The square root of 4 is 2
]RUN
Input x: -16←——user's response
The square root of -16 is 4i
```

A RESUME statement can be used at the end of an error handling subroutine to branch the program control back to the section of the program where the error occurred. The function of a RESUME statement is analogous to the use of a RETURN statement at the end of a subroutine.

8

Tables and Arrays

Introduction

In Chapter 5 we introduced the concept of variables. A variable is designed to hold a single data item — either string of numeric. However, some programs require that hundreds or even thousands of variable names be used.

The processing of large quantities of data can be greatly facilitated through the use of arrays and tables in a program.

SUBSCRIPTED VARIABLES

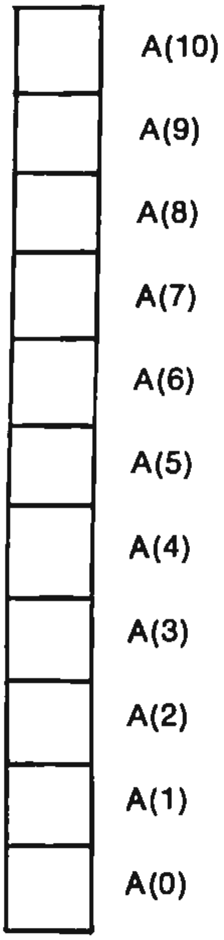
Obviously, the use of thousands of individual names could prove extremely cumbersome. To overcome this problem, BASIC allows the use of **suscripted variables.** Subscripted variables are identified with a **subscript,** a number appearing within parentheses immediately after the variable name. An example of a group of subscripted variables is given below:

```
A(0),A(1),A(2),A(3),A(4),etc.
```

Note that each subscript variable is a unique variable. In other words A(0) differs from A(1), A(2), A(3), etc. . .

Arrays and Tables

Subscripted variables may be visualized as an **array** (or **table**). In our previous example, the data contained in the array defined by A would consist of a one-dimensional array with 11 elements.



Arrays can also have two or more dimensions. Two-dimensional arrays are also known as tables. A table containing 6 rows and 8 columns is depicted on the following page.

		Columns						
Rows	0	1	2	3	4	5	6	7
0	A(0,0)	A(0,1)	A(0,2)	A(0,3)	A(0,4)	A(0,5)	A(0,6)	A(0,7)
1	A(1,0)	A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)	A(1,6)	A(1,7)
2	A(2,0)	A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)	A(2,6)	A(2,7)
3	A(3,0)	A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)	A(3,6)	A(3,7)
4	A(4,0)	A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)	A(4,6)	A(4,7)
5	A(5,0)	A(5,1)	A(5,2)	A(5,3)	A(5,4)	A(5,5)	A(5,6)	A(5,7)

Notice from our illustration that a position within the table is identified with a subscripted variable. The subscript contains two numbers. The first number identifies the row number and the second identifies the column. For instance, A(1,2) identifies the element located in column two of row one.

Array variables can be assigned values and used with operators as can any other variable. This is illustrated in the following example.

```
]10 A(0) = 5
]20 A(1) = 6
]30 A(2) = 7
]40 A(3) = 8
]50 A(4) = 9
]60 PRINT A(0) * A(1)
]70 A(5) = A(2) + A(3) + A(4)
]80 PRINT A(5)
]90 END
]RUN
30
24
```

DIMENSIONING AN ARRAY

Before an array variable can be used in a program, an area in memory must be reserved to store its elements. This is known as dimensioning the array and is accomplished with the DIM statement.

The DIM statement defines the maximum subscript value that can be used for an array. For example, the following DIM statement:

```
DIM A(20)
```

would define a one-dimensional array consisting of twenty-one elements ranging from A(0) to A(20) inclusive.

Two-dimensional arrays are dimensioned as follows:

```
DIM A(4,7)
```

The preceding DIM statement would dimension an array consisting of five rows with eight columns each.

Notice that a DIM statement was not included in our first example. When a subscripted variable which has not been previously dimensioned is referenced in a program, the array variable is automatically dimensioned with a maximum subscript value of 10. If we added the following program line to our example:

```
85 A(11) = 24:PRINT A(11) * A(1)
```

the following error message would be displayed:

```
?Bad Subscript Error in 85
```

This error is generated because an array variable was referenced with a subscript greater than orginally stated.

If the following DIM statement was inserted in our example program:

```
5 DIM A(11)
```

it would execute properly, because A(11) would have been defined by the DIM statement.

Generally, it is good programming practice to dimension all array variables and to group all DIM statements at the beginning of the program. This prevents an array variable from inadvertently being referenced before it has been dimensioned.

When an array is no longer needed in a program, the DIM statement can be reversed with a CLEAR statement. This will free the memory area previously reserved for the array. This is illustrated in the following program.

```
]10 PRINT FRE(0)
]20 DIM A(50,50)
]30 PRINT FRE(0)
]40 CLEAR
]50 PRINT FRE(0)
]60 END
]RUN
25992
12982
25992
```

In line 10, the number of available bytes in memory are displayed. FRE is a function which displays the available free bytes in memory. FRE is explained in more detail in Chapter 9.

In line 20, the DIM statement reserves an area in memory for a table consisting of 2601 elements. From line 30, it is evident that the number of free bytes has decreased substantially. This is due to the fact that an area of memory has been reserved for the elements in table A.

In line 40, the CLEAR statement reverses the DIM statement and the memory previously required for the elements in table A are freed.

DATA & READ Statements

Earlier, we discussed how data could be assigned to a variable with . LET statement as well as how data could be input directly from the keyboard and assigned to a variable with an INPUT or GET statement. However, none of these statements are practical for assigning data values to the individual variables in a large array or table. DATA and READ statements are much more practical for assigning values to variables in an array. DATA and READ statements can be used for assigning values to any variable — not just array variables.

A typical DATA statement is shown below.

```
100 DATA "WILLIAMS",27,"ST.LOUIS","314-727-1141"
```

Notice that this DATA statement contains four data items, three of which are string and one of which is numeric. In our example, we have enclosed the string data items in quotation marks. However, this was not actually required. In a DATA statement, a string only needs to be enclosed in quotation marks if it contains a comma, a colon, or if its first character is a blank space.

DATA statements are used in conjunction with READ statements to assign data values to variables. An example of a READ statement is given below.

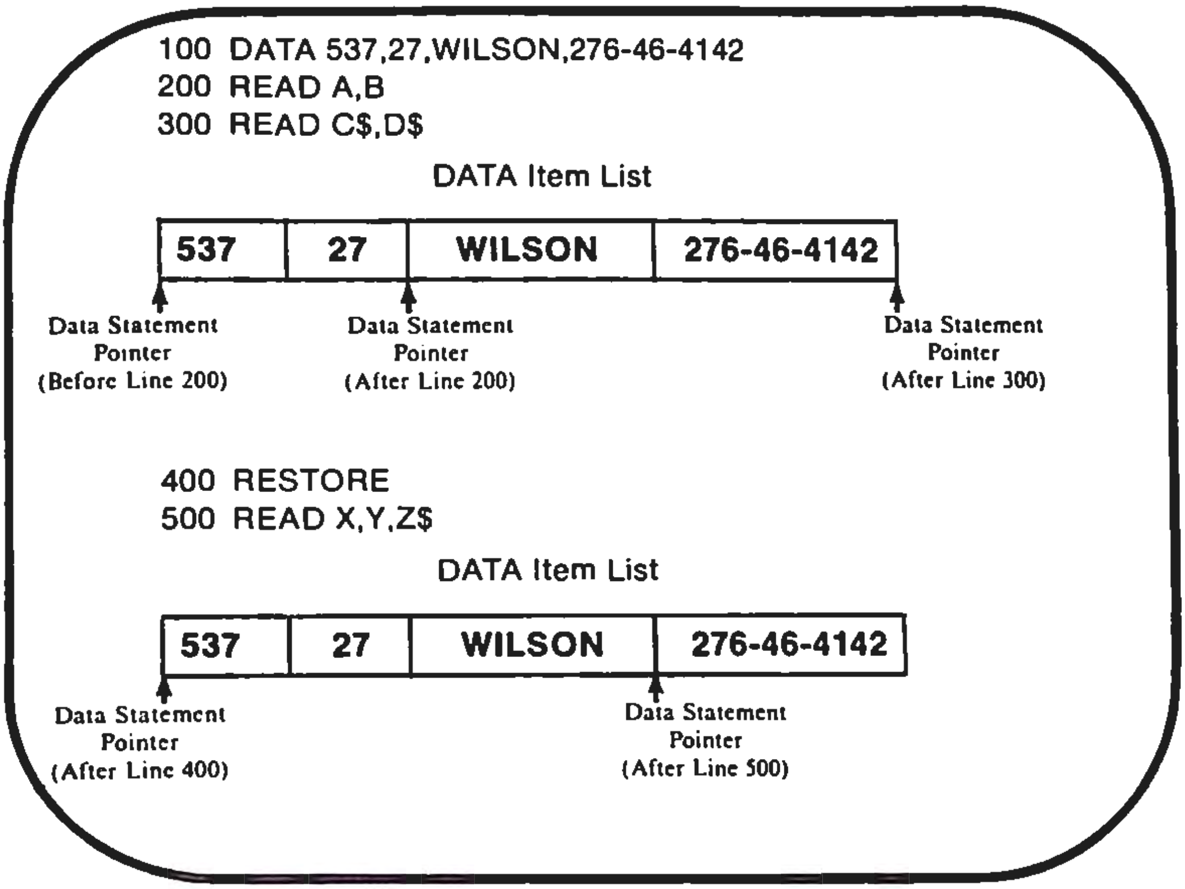
```
200 READ NAME$,AGE,CITY$,PHONE$
```

When a READ statement is executed, the computer will first search for a DATA statement. When a DATA statement is found, the values in the DATA statement will be assigned one-by-one to the variables in the READ statement.

If the first DATA statement encountered does not have enough data items to be assigned to all the variables in the READ statement, the next DATA statement will be searched for. The values from this and succeeding DATA statements will continue to be assigned to the variables in the READ statement until all of the variables in the READ statement have been assigned a value.

The computer keeps track of the next DATA statement data item to be used via an internal pointer. When any future READ statements are executed, this pointer will determine which is the next data item to be read into the READ variable.

BASIC includes a statement known as RESTORE, which when executed, sets the DATA item pointer back to the beginning of the DATA statement list. The use of the DATA item pointer and the effect of RESTORE on it is depicted in Figure 8-1.



```
100 DATA 537,27,WILSON,276-46-4142
200 READ A,B
300 READ C$,D$
```

DATA Item List
537
27
WILSON
276-46-4142
Data Statement Pointer (Before Line 200)
Data Statement Pointer (After Line 200)
Data Statement Pointer (After Line 300)

```
400 RESTORE
500 READ X,Y,Z$
```

DATA Item List
537
27
WILSON
276-46-4142
Data Statement Pointer (After Line 400)
Data Statement Pointer (After Line 500)

FIGURE 8-1. DATA Statement Pointer

When not properly used, DATA and READ statements can be the source of program errors. One potential error source occurs when the program attempts to READ more data items than were given in the DATA statements. Such an error would occur in the following program.

```
]100 DATA 7,8,11,13,15
]200 FOR K = 1 TO 7
]300 READ X(K)
]400 PRINT X(K)
]500 NEXT K
]600 END
```

In the preceding example, the program would attempt to read 7 data items. However, since the DATA statement only contained 5 data items, the following error message would appear:

```
?Out of DATA Error In 300
```

Another potential source of error when executing DATA and READ statements are situations where the program attempts to read a numeric data item into a string variable or vice versa. If such an error is encountered the following error message will be displayed:

```
?Syntax Error
```

DATA and READ statements are often used in conjunction with FOR, NEXT loops to read large amounts of data into arrays. An example of this use of FOR, NEXT is given below:

```
]10 FOR K =0 TO 5
]20 READ NAMES$(K)
]30 READ AGE(K)
]40 NEXT K
]50 FOR J = 0 TO 5
]60 PRINT NAMES$(J),AGE(J)
]70 NEXT J
]80 END
]90 DATA Jim,10
]100 DATA Tom,11
]110 DATA Matt,9
]120 DATA Eric,10
]130 DATA Steve,10
]140 DATA Joe,9
]Run
Jim                    10
Tom                    11
Matt                   9
Eric                   10
Steve                  10
Joe                    9
```

An example of the use of the READ and DATA statements in conjunction with a FOR, NEXT loop for the purpose of reading data into a two-dimensional array is given in the following program.

```
] 10 DATA 10,20,30,40
] 20 DATA 50,60,70,80
] 30 DATA 90,10,20,30
] 40 FOR J = 0 TO 2
] 50 FOR K = 0 TO 3
] 60 READ A(J,K)
] 70 PRINT A(J,K);" ";
] 80 NEXT K
] 90 PRINT
]100 NEXT J
]110 END
]RUN
10     20     30     40
50     60     70     80
90     10     20     30
```

The preceding program would read data items into the table A( ) as shown in Figure 8-2.

	0	1	2	3
0	10	20	30	40
1	50	60	70	80
2	90	10	20	30

FIGURE 8-2. A( ) Array Values

9

Functions and String Handling

Introduction

In mathematics, a function is generally defined as a quantity whose value will vary as a result of another quantity. In computing, functions define operations that are performed on stings or numeric values.

In BASIC, a number of functions are already defined by reserved words and are a part of the BASIC interpreter. These are known as **built-in** functions (see Table 9-1). Built-in functions cover a wide range of standard math operations such as absolute value, square root, logarithms, etc. Built-in functions are also available for working with strings, as well as a variety of other operations.

BASIC also allows the programmer to define his or her own functions. These are known as **user-defined** functions. Both built-in and user-defined functions will be discussed in this chapter, as well as in Chapter 12.

Built-in Numeric Functions

MATHEMATICAL FUNCTIONS

The majority of the SmartBASIC functions are used in mathematical applications. We will provide an overview of SmartBASIC's math functions in this section. Each individual function will be described in detail in Chapter 12.

Table 9-1. Smart BASIC Built-in Functions

ABS
ASC
ATN
CHR$
COS
EXP
FRE
INT
LEFT$
LEN
LOG
MID$
PDL
PEEK
POS
RIGHT$
RND
SCRN
SGN
SIN
SPC
SQR
STR$
TAB
TAN
VAL

All of the SmartBASIC mathematical functions operate in much the same manner. Each function is defined by a reserved word (ex. SIN for Sine, COS for Cosine, LOG for Logarithm, etc.).

A numeric constant, variable, or expression may appear in parentheses following the reserved word which identifies the function. The function for that numeric value will then be calculated by the computer. The use of several mathematical functions is shown in Figure 9-1.

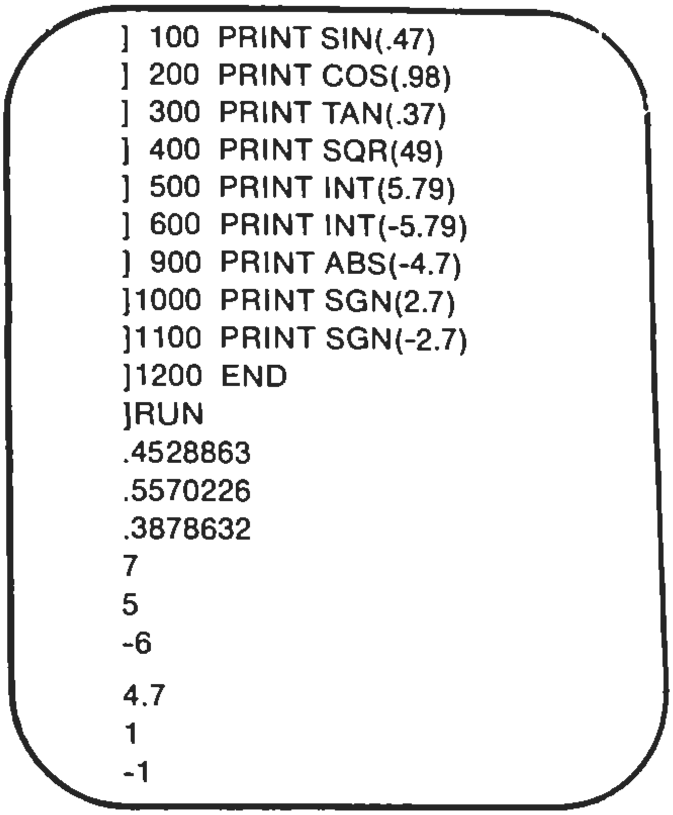
SmartBASIC includes the following three trigonometric functions:

```
SIN(N) = sine of the angle N.
COS(N) = cosine of the angle N.
TAN(N) = tangent of the angle N.
```

The angle N must be given in terms of radians. One radian is the equivalent of 57.29578 degrees. One degree equals .017453 radians.

Therefore, the following can be used to calculate a trigonometric function with its argument (X) given in degrees:

```
SIN(.017453 * X)
COS(.017453 * X)
TAN(.017453 * X)
```



```
] 100  PRINT SIN(.47)
] 200  PRINT COS(.98)
] 300  PRINT TAN(.37)
] 400  PRINT SQR(49)
] 500  PRINT INT(5.79)
] 600  PRINT INT(-5.79)
] 900  PRINT ABS(-4.7)
]1000  PRINT SGN(2.7)
]1100  PRINT SGN(-2.7)
]1200  END
]RUN
.4528863
.5570226
.3878632
7
5
-6
4.7
1
-1
```

FIGURE 9-1. Mathematical Functions

The other three principal trigonometric functions: secant, cosecant, and cotangent can be computed by using SIN, COS, and TAN as shown in the following idenities.

```
SEC(X) = 1/COS(X)
CSC(X) = 1/SIN(X)
COT(X) = COS(X)/SIN(X)
```

SmartBASIC also includes the arctangent function ATN. This function returns the angle (expressed in radians) whose tangent is given in its argument.

ATN(X) = angle in radians whose tangent equals X

The following formula can be used to calculate the angle expressed in degrees (rather than radians) whose tangent is given in X.

57.29578 * ATN(X)

BASIC also contains functions for calculating natural logarithms and exponentials. The exponential formula takes the following form:

A = EXP(B)

The preceding EXP function is calculated by computing the value of e raised to the B power. e is known as the base of natural logarithms. The value e in SmartBASIC is 2.71828183.

The natural logarithm of a number may be calculated with the LOG function.

LOG(X) = natural logarithm of X

Logarithms with a base other than e may be calculated using the following formula:

$LOG_b(X)$ = LOG(X)/LOG(b)

where b is the base of the logarithm.

SmartBASIC includes the SQR function for determining the positive square root of its argument.

SQR(X) = positive square root of X

The square root of a number can also be calculated with the exponential arithmetic operator. The following expression,

X ∧ (1/2)

will calculate the square root of X. The arithmetic exponential operator can also be used to calculate a root other than the square root (ex. cube root) as shown below.

X ∧ (1/3)

SmartBASIC also includes several functions that can be used in working with numeric values. These include INT, ABS, and SGN.

The INT function returns the integer with the greatest value which is less than or equal to its argument. INT takes the following form:

INT(X) = highest integer whose value
is less than or equal to X

Figure 9-1 contains examples of the usage of the INT function.

The ABS function returns the absolute value of its argument. ABS takes the following form.

```
ABS(X) =| x |
```

An example of the use of ABS appears in Figure 9-1.

The SGN function returns the sign of its argument. An example of the use of SGN appears in Figure 9-1.

USER-DEFINED FUNCTIONS

In the preceding section, we discussed a number of predefined SmartBASIC functions. SmartBASIC also allows the user to define his own functions. These are known as **user-defined** functions. A user-defined function must be defined with the DEF FN statement before it can be used in the program.

For example, the following DEF FN statement would define a function in which the argument was squared and 1 was then subtracted from that calculation.

```
100 DEF FN A(X) = X ^ 2-1
```

The name of the function (FN A) appears immediately following the DEF statement. Any valid variable name may be used as a user-defined function name. The following would be a valid function name:

```
FN TANH
```

In our first example, notice the X in parentheses following the function name. This is known as a **dummy argument.** Any valid variable name can be substituted for X as the dummy argument.

When the user-defined function is called in the program, the argument supplied with the function when it is called will be substituted for the dummy argument whenever it appears on the right-hand side of the DEF FN statement. The expression is then evaluated and the value is returned as the value of the function.

```
]10  M = 3.9878
]20  DEF FN S(X) = COS(X) + SIN(X)
]30  PRINT FN S(M)
]RUN
-1.41159969
```

The previous example contains a program that has a DEF FN statement at line 20. The function is assigned the name S, and the dummy argument X is used in the function. The operations in the function (COS(X) + SIN(X)) can be as complicated as necessary. At line 30, the S function is evaluated at the value of the variable M. The function substitutes 3.9878 for the dummy argument X and returns a numeric value that is displayed by the PRINT statement.

Strings & String Handling

As a programmer, you will encounter a number of situations where you may need to work with string data. For example, you might want to combine several strings, compare two strings, separate portions of a string, or even convert string data to its numeric equivalent. SmartBASIC allows for all of these.

STRING CONCATENATION

The process of joining together one or more strings is known as **concatenation.** The arithmetic operator for addition (+) is used for string concatenation. However, concatenation is very different from addition. In concatenation, the strings being concatenated are joined to form a new string as shown below.

```
]100  A$ = "JOHN"
]200  B$ = "SON"
]300  C$ = A$ + B$
]400  PRINT C$
]500  END
]RUN
JOHNSON
```

Either string constants or variables may be concatenated. Any number of strings may be concatenated as long as the resulting string contains 255 or fewer characters.

The same relational operators are used for comparing strings as are used for comparing numeric data. These include the following:

<	→	less than
<=	→	less than or equal to
>	→	greater than
>=	→	greater than or equal to
=	→	equal to
<>	→	not equal

Strings are compared one character at a time beginning with each string's first character. This comparison is made with each character's corresponding ASCII code.

Fortunately, ASCII code comparisons are relatively simple. A comparison of the characters by ASCII codes is almost identical to an alphabetical comparison. A character is less than another with respect to ASCII codes, if that character precedes it in the alphabet. Lowercase letters are always greater than their uppercase counterparts and digits are always less than letters.

Like numeric relational operators, string relational operators return a value of 0 if the relation is false and a value of 1 if it is true. Whenever strings are being compared they must be enclosed within quotation marks.

STRING HANDLING FUNCTIONS

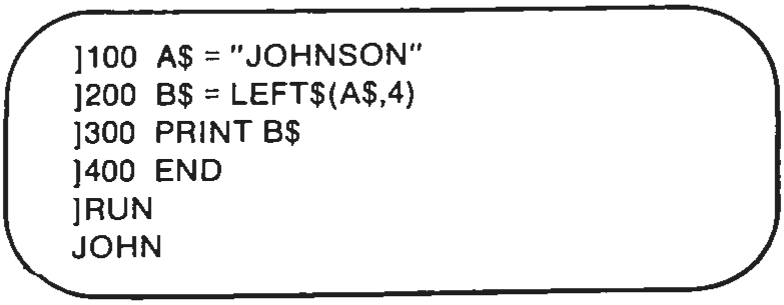
SmartBASIC contains a number of string handling functions whic allow the user to extract a part of a string. These functions are LEFT$, MID$ and RIGHT$.

The LEFT$ function takes the following format:

LEFT$(*string,X*)

where *string* is the string on which the operation is to be performed and *X* is the number of characters to be extracted. The LEFT$ function will extract the leftmost number of characters given in *X* from the string given in *string*.

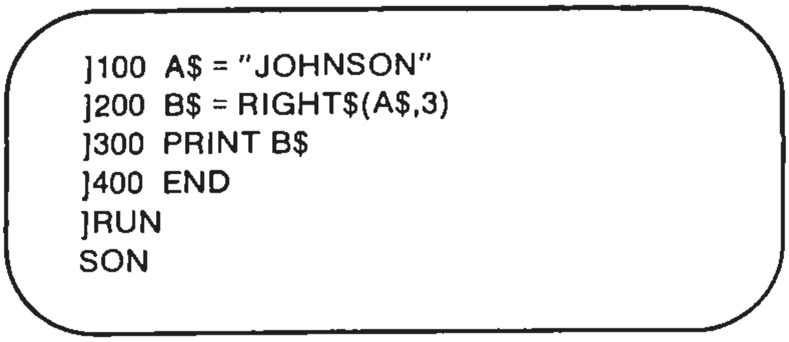
Figure 9-2 contains an example of the use of LEFT$



```
]100  A$ = "JOHNSON"
]200  B$ = LEFT$(A$,4)
]300  PRINT B$
]400  END
]RUN
JOHN
```

FIGURE 9-2. LEFT$

The RIGHT$ function works exactly like the LEFT$ function except that the rightmost number of characters specified are returned. Figure 9-3 contains an example of the use of RIGHT$.



```
]100  A$ = "JOHNSON"
]200  B$ = RIGHT$(A$,3)
]300  PRINT B$
]400  END
]RUN
SON
```

FIGURE 9-3. RIGHT$

The MID$ function can be used to return a portion of a string. MID$ takes the following format:

a$ = MID$(*b*$,*x*[,*y*])

The string being returned is *a*$. *a*$ is being returned from *b*$. The string being returned will begin with the *x*th character in *b*$. The number of characters returned from *b*$ is specified in *y*.*y* is an optional parameter. If *y* is omitted, all rightmost characters in *b*$ will be returned in *a*$. An example of the use of MID$ to return a portion of a string is given in Figure 9-4.

```
]100  X$ = "NEW CASTLE"
]200  Y$ = MID$(X$,5,4)
]300  PRINT Y$
]400  END
]RUN
CAST
```

FIGURE 9-4. MID$

STRING/NUMERIC DATA CONVERSION

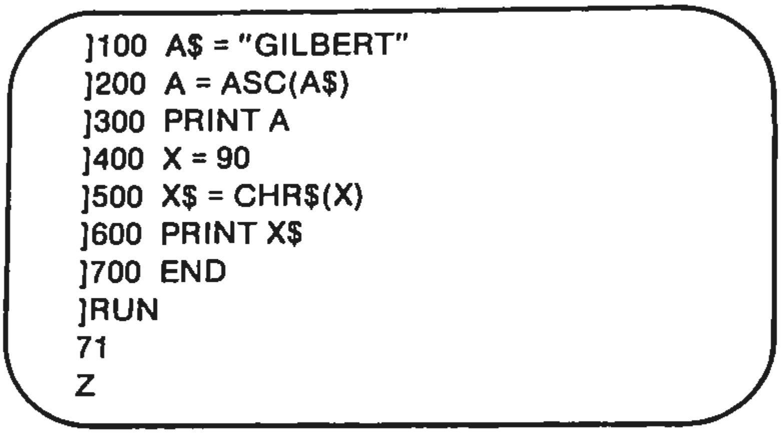
Programmers often encounter situations where numeric data must be converted into string data and vice versa. This is often the case where a function is being used which will accept only string or numeric data as its arguments.

The STR$ and VAL functions are used to convert numeric data to its string equivalent and strings to their numeric equivalent repectively. The ASC function is used to convert a single character to its ASCII numeric equivalent. If ASC is given a string it will return the ASCII equivalent of the first character in that string. The CHR$ function converts an ASCII numeric code to an equivalent text character.

Examples of the use of STR$, VAL, CHR$, and ASC are given in Figures 9-5 and 9-6.

```
]100  w = 33578
]200  w$ = STR$(w): REM w$ = "33578"
]300  x = 33579
]400  x$ = STR$(x)
]500  y$ = w$ + x$: REM y$ = "3357833579"
]600  y = VAL(y$) : REM y = 3357833579
]700  z = INT(y/10000)
]800  PRINT z
]900  END
]RUN
335783
```

FIGURE 9-5. STR$ and VAL Example



```
]100 A$ = "GILBERT"
]200 A = ASC(A$)
]300 PRINT A
]400 X = 90
]500 X$ = CHR$(X)
]600 PRINT X$
]700 END
]RUN
71
Z
```

FIGURE 9-6. CHR$ and ASC Examples

VARIABLE TABLE AND STRING STORAGE

BASIC maintains an area in memory in which an entry is maintained or every variable (including array variables) referenced either in a program or in the direct mode. This memory area is known as the **variable table.**

For numeric variables, the value currently assigned to that variable is also stored in the variable table. When that variable's value is changed, the value stored in the variable table will also be changed.

In SmartBASIC, the amount of memory required to store a numeric value in the variable table remains constant. On the other hand, the amount of memory required to store a string variable's value can vary depending upon that value.

Since the memory space required to store a string value can vary, it would be difficult to store these values in the variable table, as that table would have to be continually revised as different values were assigned to string variables. For this reason, SmartBASIC stores string values in a separate memory area known as **string space.**

SmartBASIC stores a value in the variable table which associates the string variable name (in the variable table) with its associated value in string space. This is known as the **descriptor.** The descriptor describes the number of characters currently assigned to the string variable as well as its location in string space.

Descriptors are not limited to referencing string values stored in the string space. Descriptors can reference strings stored anywhere within BASIC's working area — including file buffers and the program itself.

When a string constant is assigned to a string variable in a BASIC program (ex. 10 A$ = "JOHN"), that constant need not be stored in the string space as the descriptor can reference it in the program storage area.

HOUSEKEEPING AND FRE

Areas assigned to strings can become unused because strings in BASIC can have variable lengths. Every time a different value is assigned to a string variable, its length may change. This may cause the space assigned to a string to become partially unused. If the string space is in need of a housekeeping SmartBASIC will automatically halt program execution and perform one. This operation may be time consuming and confuse the user.

The FRE function allows the programmer to determine the amount of available memory and perform a housekeeping at the same time. The FRE function requires an argument, but the value of the argument has no effect on the operation of the function or the value returned. A tyical FRE function call would appear in a program as follows.

```
100 PRINT FRE(0)
```

10

Files and File Handling with SmartBASIC

Introduction

The Digital Data Drive supplied with the ADAM allows programs and data to be easily stored on Digital Data Packs. This feature adds a great deal of versatility to the ADAM, since the program and data stored in its memory are erased each time the computer is turned off. The Digital Data Drive allows the creation of "permanent" copies of programs and data.

FILE TYPES

Programs and data are stored on a Data Pack in units called **files.** Files consist of a single program or an organized collection of data items. Files are divided into two categories: program files and data files. As their names suggest, program files can only accommodate a program, while data files can hold only data.

FILENAMES

SmartBASIC requires that each file be assigned an appropriate name. This allows the files to be easily distinguished. Filenames can consist of 1 to 10 characters. Generally, a program's filename should provide some clue as to the actual contents of the files. The following are all acceptable filenames.

```
ACCOUNT27
FORECAST
BUDGET
ADDRESSES
```

Filenames can include uppercase letters, lowercase letters, and digits as well as many special characters including the following:

```
|!@#$%.&'-+   {
```

Each file on a Data Pack must have a unique filename. When an attempt is made to store a file with the name of a file that already exists on a Data Pack, that file will be saved in place of the original file. The original file will be placed in a backup file that can only be accessed by the word processor.

Program Files

SAVING PROGRAMS

The procedure used to save a program is very simple and straightforward. When a program is present in the computer's memory, it can be saved on a Data Pack with the SAVE command. The SAVE command must be followed by an appropriate filename. The following example statements demonstrate the correct format for the SAVE command.

```
]SAVE homework
]SAVE FORECAST28
```

Once a SAVE command has been issued, the Data Drive will operate for about a minute. Once Data Drive operation has ceased, the prompt will once again appear on the display.

LOADING PROGRAMS

Programs that have been previously saved on a Data Pack can be reloaded into the computer's memory through the use of a LOAD command. LOAD commands are analogous to the SAVE command in that only the program filename needs to be included in the command.

The following example statements demonstrate the use of a LOAD command.

```
]LOAD homework
]LOAD FORECAST28
```

When a LOAD command specifies a program file that does not exist on a Data Pack, the following message will be displayed.

```
File Not Found
```

When the LOAD command is used to recover a previously saved program, the program that resides in the computer's memory will be erased. In other words, the use of the LOAD command implies that the NEW command will be executed before the program is loaded from the Data Pack.

RUNNING PROGRAMS

The LOAD command can be used to recover a previously stored program, but the RUN command is still required to execute the program. This two-step procedure can be replaced by a single RUN command if a filename is specified. This concept can be clarified using the following example.

```
]LOAD PROGRAM
]RUN

]RUN PROGRAM
```

The last statement in the preceding example causes the same effect as the combination of the first two. Like the LOAD command, the RUN command causes the computer's memory to be cleared before the new program is loaded. The RUN command can be included in a program to cause the computer to load and execute another program.

CATALOGING A DATA PACK

As files are saved on a Data Pack, a list of filenames will be maintained by the computer. The CATALOG command can be used to display a directory of the files currently saved on a Data Pack. When the CATALOG command is executed, the Data Drive may operate for a few moments before the directory is displayed. A typical directory might appear as follows.

```
Volume: FIRST DIR
  A     15  FORECAST
  A     10  homework
  A      2  Program.A
226 Blocks Free
```

The first column of the directory contains an uppercase letter A to indicate a SmartBASIC program file. The second column of the directory indicates the number of blocks that the program occupies. A block is a unit of storage space on a Data Pack. A lowercase A indicates a backup file.

The third column of the directory contains the names of each file stored on the Data Pack. The filenames are recorded in the directory exactly as they are specified in a SAVE statement.

RENAMING A FILE

The name of any file stored on a Data Pack can be easily changed. The RENAME command is used for this function. When renaming a file, simply enter the RENAME command followed by the current name of the file and finally, the new name of the file. Be sure to separate the old and new names with a comma. The following example statement would cause the file currently named OLD to be renamed as NEW.

```
]RENAME OLD,NEW
```

If the RENAME is used to specify a new filename that already exists on the Data Pack, the RENAME command will have no effect.

DELETING A FILE

The DELETE command can be used to erase files that exist on a Data Pack. The DELETE command need only be followed by an appropriate filename. For example, the following command could be used to delete a file named OLD.

```
]DELETE OLD
```

PROTECTING A FILE

The LOCK command can be used to prevent a file from being accidentally deleted. Once a file has been locked, it can still be loaded, resaved and renamed, but it cannot be deleted. An asterisk is displayed in the directory adjacent to the name of the file.

The UNLOCK command can be used to remove the file's protection. The following series of commands illustrates the use of file protection.

```
]SAVE PROGRAM
]LOCK PROGRAM
]CATALOG
Volume: FIRST DIR
*A     10     PROGRAM
243 Blocks Free
]DELETE PROGRAM
File Not Found
]UNLOCK PROGRAM
]DELETE PROGRAM
]LOAD PROGRAM
File Not Found
```

Note that once the file has been saved and locked, the filename will appear in the directory with an asterisk. The file cannot be deleted until it has first been unlocked. The final LOAD command indicates that the file has actually been deleted.

Initializing a Data Pack

All of the files stored on a data pack can be deleted with a single command. The INIT command can be used to erase the contents of the directory and to establish a volume name. The volume name is the title that appears on the display each time the directory is accessed. The desired volume name must be included in the INIT command. For example, the following INIT statement clears the directory and sets the volume name to FINANCIAL.

```
]INIT FINANCIAL
]CATALOG
Volume: FINANCIAL
253 Blocks Free
```

Notice that the INIT command causes the previous files of a Data Pack to be erased.

USING DATA DRIVE COMMANDS IN PROGRAMS

The commands used to manipulate files can be included in SmartBASIC programs through PRINT statements. As a result, files can be saved, loaded, renamed, deleted, locked or unlocked during the execution of a program.

A special character code can be used to indicate that a PRINT statement contains a Data Drive command. The special character can be generated with the CHR$ function as follows:

```
CHR$(4)
```

When the special character is used in a PRINT statement, the subsequent data is considered to be a Data Drive command. For example, the following statement would cause the file named PROGRAM to be deleted.

```
PRINT CHR(4);"DELETE PROGRAM"
```

It is usually convenient to define a string variable which references the special character. For example, the following statement could be used to simplify this process.

```
D$ = CHR$(4)
```

This statement allows the following PRINT statements to perform Data Drive commands.

```
PRINT D$;"SAVE PROGRAM"
PRINT D$;"CATALOG"
PRINT D$;"DELETE PROGRAM"
PRINT D$;"RUN PROGRAM"
```

Data Files

Data files provide a convenient means of storing information that can later be used in programs. Data files may contain any number of numeric or string values. These data items are "written" into the file in a sequential manner, and must be "read" from the file in the same manner.

A data file can be easily manipulated by SmartBASIC programs through the use of PRINT statements containing Data Drive commands. The most commonly used Data Drive commands for data files are OPEN, CLOSE, READ and WRITE.

OPENING AND CLOSING FILES

Before a file can be accessed, it must first be opened. As a result, an OPEN command must appear in a program before any other data file commands. An OPEN command must include the name of the file to be opened.

A file does not need to exist before a program actually accesses the file. In other words, if an OPEN statement specifies a file that does not exist, a new file will be created.

Since each active file requires an allocation of the computer's memory, it is recommended that files be closed when they are not in use. As expected, a CLOSE statement can be used to close a previously opened file. For a particular data file called EXPENSES, the OPEN and CLOSE commands would have the following structure.

```
PRINT CHR$(4);"OPEN EXPENSES"
PRINT CHR$(4);"CLOSE EXPENSES"
```

A CLOSE command is not an essential aspect of data file handling because each file is automatically closed when a SmartBASIC program ends.

READING AND WRITING DATA

Data is written to a file in essentially the same manner it is written to the display. However, a WRITE command must be issued before the data can be recorded on a Data Pack. Once a WRITE command has been issued, any subsequent PRINT statement will cause data to be output to the file instead of to the display. Consider the following example program which outputs ten numeric values, to a data file.

```
]10 d$ = CHR$(4)
]20 PRINT d$; "OPEN OUTPUT"
]30 PRINT d$; "WRITE OUTPUT"
]40 FOR t = 1 TO 10
]50 PRINT t
]60 NEXT t
]70 PRINT d$; "CLOSE OUTPUT"
]80 END
```

The assignment statement at line 10 causes the string variable d$ to be assigned the value of the special character, CHR$(4). Line 20 establishes the data file and Line 30 indicates that the subsequent PRINT statements will cause data to be sent to the file. The loop from lines 40 to 60 causes the data to be output before the file is closed at line 70.

The method used to input data from a file is very similar to that used to input data from the keyboard. However, a READ command must be issued before data can be accepted from a file. Once a READ command has been executed, data can be retrieved from a file through the use of INPUT statements. It is usually convenient to include a prompt message in an INPUT statement to supress the question mark which is ordinarily generated.

Before data can be read from a file, the data must first have been saved in an appropriate format. The previous example program can be modified is such a way as to recover the data from the file OUTPUT.

```
]10 d$ =CHR$(4)
]20 PRINT d$;"READ OUTPUT"
]30 PRINT d$;"READ OUTPUT"
]40 FOR t = 1 TO 10
]50 INPUT "";x: PRINT x
]60 NEXT t
]70 PRINT d$;"CLOSE OUTPUT"
]80 END
```

Notice that only lines 30 and 50 have been altered. These lines indicate that the data will be input rather then output. As a result, this program recovers the data stored in the file OUTPUT.

The storage of data items in data files must be performed carefully. When values are written to a data file, each individual value must be properly set apart from the rest of the data in the file. A special character called a Carriage Return (CR) must be used to separate the data items.

The PRINT statements used to write data to a file are analogous to the statements used to output data to the display.

Recall from the discussion of the PRINT statement in Chapter 6 that a CR is generated at the end of each PRINT statement unless the statement ends with a comma or semicolon. As a result, the easiest means of outputting data to a file is to include each data item in its own PRINT statement. The following program demonstrates this principle. Notice that only variable (x$) appears in the PRINT statement.

```
]100 d$ = CHR$(4)
]200 PRINT d$;"OPEN TEXT"
]300 PRINT d$; "WRITE TEXT"
]400 INPUT "";x$
]500 PRINT x$
]600 IF x$ = "END" THEN 800
]700 GOTO 400
]800 PRINT d$; "CLOSE TEXT"
]900 END
```

When the preceding example program is executed, a data file named TEXT will be established Data entered at the keyboard will be written to the file until the work END has been entered. This value, called a flag value, determines the end of the data entry procedure.

The data can be recovered from the file by editing line 300 as follows.

```
300 PRINT d$;"READ TEXT"
```

More than one data item can be included in a PRINT statement, as long as each individual data items is separated with the CR character. This is illustrated in the following example. Note the usage of CHR$(13) to denote CR.

```
]100 d$ = CHR$(4)
]200 INPUT "name? ";n$
]300 INPUT "address? ";a$
]400 INPUT "phone? "p$
]500 PRINT d$;"OPEN TEXT"
]600 PRINT d$;"WRITE TEXT"
]700 PRINT n$; CHR$(13); a$; CHR$(13); p$
]800 PRINT d$; "CLOSE TEXT"
]900 END
```

The following program can be used to recover the data stored in the file created in our preceding example. The HOME statement at line 500 display to be cleared before the output appears.

```
]100 d$ = CHR$(4)
]200 PRINT d$; "OPEN TEXT"
]300 PRINT d$; "READ TEXT"
]400 INPUT ""; n$
]500 INPUT ""; a$
]600 INPUT ""; p$
]700 PRINT n$; CHR$(13); a$; CHR$(13); p$
]800 PRINT d$; "CLOSE TEXT"
]900 END
```

The I/O Monitor

The data being transferred between the computer and Data Drives can be displayed on the monitor as that transfer occurs. The MON and NOMON commands can be used to activate and deactivate the monitor.

The monitor can be used to display data being output, data being input, and/or Data Drive commands. The letters C, I and O are used to select the command monitor, input monitor and output monitor. Table 10-1 summarizes the modes of the I/O monitor.

Table 10-1. MON Command Summary

MON Command	Data Monitored
MON C,I,O	Commands, Input and Output
MON C,I	Commands and Input
MON C,O	Commands and Output
MON I,O	Input and Output
MON O	Output
MON I	Input
MON C	Commands

The following program demonstrates the use of the I/O monitor. Notice that the NOMON cancels the MON command.

```
]100 d$ = CHR$(4)
]110 PRINT d$;"MON C,I,O"
]120 PRINT d$;"OPEN FILE"
]130 PRINT d$;"WRITE FILE"
]140 FOR j = 1 TO 5
]150 PRINT j
]160 NEXT j
]170 PRINT d$;"CLOSE FILE"
]180 PRINT d$;"NOMON C,I,O"
]190 PRINT d$;"DELETE FILE"
]RUN
MON C,I,O
OPEN FILE
WRITE FILE
1
2
3
4
5
♥CLOSE FILE
NOMON C,I,O
```

PROBLEMS WITH DATA FILES

Occasionally, a programming error may occur in which extremely large files may be accidentally created. When this situation occurs, a No More Room error may occur even after these troublesome files have been deleted. In this situation, data files will not be allowed on the particular Data Pack until it is initialized.* If this procedure becomes necessary, be sure that you have copies of all your important files, then use the INIT command. Keep in mind that the INIT command eliminates every file on a Data Pack.

* A SmartBASIC Data Pack cannot be initialized.

11

ADAM Graphics

Introduction

The Coleco ADAM has three different display modes, one text mode and two graphics modes. These modes may be combined into four different display formats. Both of the graphics modes are color capable. These two modes will be discussed in this chapter.

In its high resolution mode, the ADAM can display 14 colors with a screen resolution of 256 x 192 pixels.* Although most competitively priced home computers can support a screen resolution comparable to the ADAM's, none can display 14 colors in a high resolution mode. In fact, most can display only 2-6 colors at high resolution. The Coleco is outstanding at producing stationary screen images. As advanced as the ADAM's color hardware is, it lacks the hardware to smoothly move images across the screen. This limitation is apparent when the movement of large objects is attempted.

* A pixel can be defined as a single screen coordinate.

Low Resolution Graphics

The low resolution graphics mode combines a graphics display with four lines of text display. The graphics display has a resolution of 40 x 40 pixels and is capable of displaying 16 colors. The four lines of text are located directly beneath the graphics display.

COMMANDS

The GR command is used to call the low resolution graphics mode. This command configures the computer hardware to display the graphics plus text format. This command also clears the display memory so that the screen will initially be black. The GR command can be used in a program or executed directly from the keyboard, as can all graphics commands.

```
GR        set low resolution
          graphics & text
```

Besides clearing the screen, the GR command also sets the low resolution "next color" register to 0. This register stores the numeric value of the next color to be displayed.

Before any graphics information can be plotted on the screen, a color must be selected. This is accomplished through use of the COLOR command. The correct syntax of this command is as follows:

```
COLOR = x
```

x represents a color number between 0 and 15. If *x* is not in this range, an "Illegal Quantity Error" results. The colors and their associated numbers are listed in Table 11-1. Although SmartBASIC is similar to Applesoft BASIC, the two execute the COLOR command differently. Applesoft allows *x* to range from 0 to 255. This point should be noted when transposing programs written in Applesoft into SmartBASIC.

TABLE 11-1. Low Resolution Color Numbers

0- black	8- yellow
1- magenta	9- medium red
2- dark blue	10- grey
3- dark red	11- pink
4- dark green	12- light green
5- grey	13- light yellow
6- medium blue	14- cyan
7- light blue	15- white

After a COLOR has been selected, information can be plotted to the screen. This is accomplished by using the PLOT command. The correct syntax of this command is as follows:

PLOT *x*,*y*

x is the column number. *y* is the row number. The column numbers extend from 0 (left) to 39 (right). The row numbers extend from 0 (top) to 39 (bottom). For example, the following program randomly plots small squares on the screen. For a nice visual effect, allow the program to run awhile. To stop the program, press CONTROL-C.

```
]10 GR
]20 COLOR = 16*RND(1)
]30 PLOT 40*RND(1),40*RND(1)
]40 GOTO 20
```

VLIN and HLIN can be used to plot consecutive pixels. VLIN [illegible] HLIN are abbreviations for Vertical LINe and Horizontal LINe, respectively. The correct syntax for the VLIN command is as follows:

VLIN y_1,y_2 AT *x*

y_1 and y_2 represent the range of row numbers. *x* represents the column number. The following VLIN command will plot every pixel in column 30 from row 4 to row 24.

VLIN 4,24 AT 30

The correct syntax for an HLIN command is as follows:

```
HLIN x1,x2 AT y
```

x_1 and x_2 represent the range of column numbers. *y* represents the row number. The following HLIN command will plot every pixel in row 14 from column 2 to column 37.

```
HLIN 2,37 AT 14
```

After information has been output to the screen, it may become necessary to determine which color is displayed at a certain screen position. The function SCRN takes as its arguments the row and column numbers, and returns the color number. The correct syntax of the SCRN command is as follows:

```
X = SCRN (x,y)
```

x represents the column number. *y* represents the row number. Upon execution of the preceding command, X will be assigned the value of the color at screen location column =*x*, row =*y*.

Use of Low Resolution Graphics

The use of low resolution graphics is usually limited to drawings and simple charts. However, the BASIC programmer may elect to use low resolution graphics as a trade-off to increase program speed. The speed of low resolution graphics, as opposed to high resolution graphics, will be utilized when the game, "BARACADE", is designed for the ADAM, later in this chapter.

CHARTS

Simple bar charts are easily implemented by using low resolution graphics. The text window may be used for documentation or labels. For example, the following program displays an annual sales chart. Line 10 clears the screen and sets up low resolution graphics. Lines 20-30 print the documentation. Lines 40-110 draw the chart using random figures.

```
] 10 GR:HOME
] 20 PRINT "WIDGET INC. STOCKS"
] 30 PRINT "RED = 1983    GREEN = 1984"
] 40 FOR I = 6 TO 35
] 50 G = 15*RND(1)
] 60 R =15 + 15*RND(1)
] 70 COLOR = 12
] 80 VLIN G,39 AT I
] 90 COLOR = 3
]100 VLIN R,39 AT I
]110 NEXT I
```

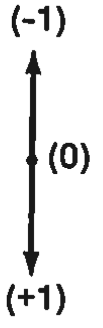
Writing a Game Program

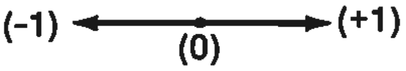
In this section, the game, "BARACADE" will be designed. The object of the game is to avoid the baracades, your own trail, and your opponent's trail. The game will be written in BASIC so that it may easily be modified. If the reader does not wish to follow the step by step designing of BARACADE, he may page through the chapter. All program lines may easily be distinguished from the rest of the text. To play BARACADE, merely enter every line belonging to the program.

The first step in designing "BARACADE" is to program the computer to draw a trail. The following statements accomplish this.

```
] 10 GR:HOME
] 70 X(0) = 15 : Y(0) = 20
] 80 DX(0) = 1 : DY(0) = 0
]110 TEMP = PDL(5 - I)
]130 IF TEMP = 1 THEN DY(I) = -1 : DX(I) = 0
]140 IF TEMP = 2 THEN DY(I) = 0 : DX(I) = 1
]150 IF TEMP = 4 THEN DY(I) = 1 : DX(I) = 0
]160 IF TEMP = 8 THEN DY(I) = 0 : DX(I) = -1
]170 X(I) = X(I) + DX(I)
]180 Y(I) = Y(I) + DY(I)
]200 COLOR = I + 3
]210 PLOT X(I),Y(I)
]230 GOTO 110
```

Line 10 clears the screen and enables low resolution graphics. Line 70 sets the initial position at screen location (15,20). The DX and DY in line 80 are the direction variables.

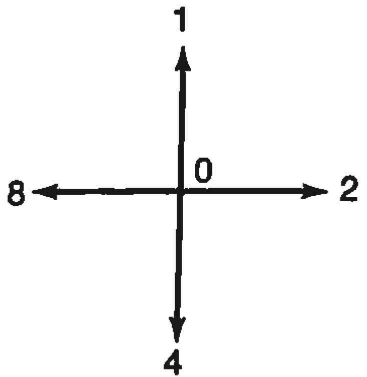




DX

DY

Initially, the direction of movement is set to the right. Lines 110-230 set up a loop which monitors joystick #1 and acts accordingly. Since I = 0 everywhere in this program, the variable TEMP is assigned the value of PDL(5). The value returned corresponds to the position of joystick #1. PDL(4) will later be used with joystick #2.



Lines 130-160 recalculate the direction variables based on the joystick position. Lines 170-180 recalculate the current position variables, and Line 210 plots the new position. Executing the program is the best way to understand how it operates.

The program has been written so as to simplify the addition of a second player. Array variables were used so that the same loop can control both players. By including the following three lines in the program, a second player can enjoy BARACADE:

```
] 90  X(1) = 25 : Y(1) = 20
]100  DX(1) = -1 : DY(1) = 0
]220  I = NOT I
```

Lines 90-100 set the initial position and direction of the second player. Line 220 alternates between selecting player #1 (I = 0) and player #2 (I = 1).

Recall from our description of "BARACADE," that the purpose of the game was for the players to avoid colliding with any barriers. The SCRN function will be used to check for collisions. If the following line is added to the program, collisions will be detected.

```
]190 IF SCRN (X(I),Y(I)) > 0 THEN 240
```

The program has not yet been completed. When it is run, an error will occur after every collision. This is because the computer has not yet been instructed what to do upon collision. Let's tell it by adding the following lines to the program.

```
]240 PRINT "COLLISION"
]250 FOR J = 1 TO 5
]260 PRINT CHR$(7);:NEXT J
]270 IF I = 1 THEN PRINT "RED WINS"
]280 IF I = 0 THEN PRINT "GREEN WINS"
]290 IF PDL(6) AND PDL(7) THEN 10
]300 GOTO 290
```

The PRINT statement in line 260 activates the television speaker. Lines 270-280 determine the winning player. Lines 290-300 delay the computer until both players are ready for another game. Pressing the left button on the controller indicates that a player is ready.

A playing field can be added by using the following lines.

```
]20 COLOR = 1
]30 VLIN 0,39 AT 0
]40 VLIN 0,39 AT 39
]50 HLIN 0,39 AT 0
]60 HLIN 0,39 AT 38
```

The program as it stands has one minor bug. If a player tries to change direction by 180°, he will lose. This is because, as far as the computer is concerned, the player ran into himself. Although this does not detract from game play, it can be annoying. When the following line is added to the program, the bug will be corrected.

```
]120 IF (TEMP = 1 AND DY(I) = 1) OR
        (TEMP = 2 AND DX(I) = -1) OR
        (TEMP = 4 AND DY(I) = -1) OR
        (TEMP = 8 AND DX(I) = 1) THEN 170
```

The ideas in this section by no means exhaust the possibilities that could be added to "BARACADE". Other upgrades might include: keeping track of games won, adding a more complex playing field, or making one player faster than the other. The only two limiting factors are execution speed and one's imagination.

High Resolution Graphics

The ADAM can also be configured to display one of two high resolution graphics formats. A graphics-only display is available that has a resolution of 256 x 192 pixels. Also, a graphics plus text display is available that has a resolution of 256 x 160 pixels. In this format, four lines of text are located beneath the graphics display.

COMMANDS

All high resolution commands directly parallel their low resolution .ounterparts. Therefore, familiarity with the low resolution commands will be assumed throughout the remainder of this chapter.

The HGR command can be used to configure the computer to display the graphics plus text format. This command clears the display memory so that the screen will initially be black.

```
HGR     set high-resolution
        graphics + text
```

The HGR command clears the high resolution "next color" register.

As previously mentioned, the ADAM can also be configured to display a graphics-only format. The HGR2 command configures the ADAM to the graphics-only high-resolution format. All high resolution graphics commands operate in an identical fashion, whether used with HGR or HGR2.

```
HGR2    set high-resolution
        graphics only
```

An HCOLOR command must be used before any graphics may be output to the high resolution display. This command selects the color that will be displayed next. The correct syntax of this command is as follows:

HCOLOR = *x*

x represents a number between 0 and 15. Each number corresponds to a specific color. This information is contained in Table 11-2.

TABLE 11-2. High Resolution Color Numbers

0- black	8- brown
1- green	9- dark blue
2- violet	10- grey
3- white	11- pink
4- black	12- dark green
5- orange	13- yellow
6- blue	14- aqua
7- white	15- magenta

High resolution graphics has a single command that can be used to output data to the screen. HPLOT is more flexible than PLOT, HLIN, and VLIN combined. In HPLOT's simplest form, it functions as low resolution's PLOT command.

HPLOT *x*,*y*

x is the column number. *y* is the row number. The column numbers extended from 0 to 255. The row numbers extend from 0 to 191. The computer stores the column number and row number of the last plotted coordinate.

The next form of the command is as follows:

HPLOT TO *x*,*y*

The computer executes this command by drawing a straight line from the last point to the point with coordinates (x,y). The line color will be the last color selected by the HCOLOR command. These two versions of the HPLOT command may be combined as follows:

HPLOT x_1,y_1 TO x_2,y_2

This command will cause a line to be drawn from screen coordinate (x_1y_1) to screen coordinate (x_2,y_2). x_2 and y_2 will be stored as the last plotted coordinate.

In order to erase a previously displayed shape, the ROTation, SCALE and position variables must be the same as when the shape was drawn. Effectively, XDRAW performs a DRAW using 0 (black) as the color variable regardless of the selected high resolution color.

Programming Using High Resolution Graphics

The following program is an example of drawing using high resolution graphics.

```
] 10 HGR2
] 20 HCOLOR = 11
] 30 HPLOT 128,31
] 40 FOR I = 0 TO 12 STEP .2
] 50 HPLOT TO 128 + 65*SIN(I),96-65*COS(I)
] 60 NEXT I
] 70 HCOLOR = 8
] 80 FOR I = 1 TO 100
] 90 HPLOT TO 64 + 128*RND(1),24 + 48*RND(1)
]100 NEXT I
]110 SCALE = 5
]120 ROT = 0
]130 HCOLOR = 9
]140 DRAW 1 AT 108,90
]150 DRAW 1 AT 148,90
]160 ROT = 8
]170 SCALE = 2
]180 DRAW 1 AT 128,110
]190 HCOLOR = 2
]200 HPLOT 98,120 TO 108,140
]210 HPLOT TO 148,140 TO 158,120
```

Line 10 enables high resolution full screen graphics. Lines 20, 70, 130 and 190 set the color register. Lines 30-60 draw a circle. Lines 80-100 draw 100 random lines. Lines 110, 120, 160 and 170 set the size and rotation of the squares that are drawn. Lines 140, 150 and 180 draw the squares. Lines 200 and 210 draw the mouth.

12

SmartBASIC Reference Guide

Introduction

In this chapter, we will provide descriptions of the various commands, statements, and functions used in SmartBASIC.

The following rules and abbreviations will be followed in this chapter in our configuration descriptions of the various BASIC commands, statements, and functions.

1. Any capitalized words are keywords.
2. Any words, phrases, or letters shown in lowercase italics identify an entry that must be made by the operator (unless enclosed within brackets).
3. Any items enclosed in brackets [ ] are optional.
4. An ellipsis (...) shows that an item may be repeated as often as desired.
5. Any punctuation marks, except the square brackets (ex. ; , =) must be included where they are shown.

ABS

The ABS function returns the absolute value of the *argument*. A number's absolute value is its value without regard to sign.

Configuration

ABS(*argument*)

The *argument* can be any numeric expression or numeric constant.

Example

```
]10 A = ABS(-1 * 7)
]20 PRINT A, ABS(2.99)
]RUN
7               2.99
```

In the preceding example, the absolute values of -7 and 2.99 are returned.

AND

AND is a logical operator. This reserved word is generally used to compare two logical expressions in the context of an IF, THEN statement.

Configuration

expression1 AND *expression2*

expression1 and *expression2* are Boolean expressions. If an *expression* was numeric (not zero), that *expression* would evaluate as true. For example, if an expression evaluated to 5, AND would treat it as true. The following is the truth table for AND.

X	Y	X AND Y
true	true	true
true	false	false
false	true	false
false	false	false

In SmartBASIC, a true is represented by a 1 and false by a 0.

Example 1

```
]10 A = 2
]20 B = 3
]30 IF (A = 2) AND (B = 3) THEN 60
]40 PRINT "AND FAILED LOGICAL TEST"
]50 GOTO 70
]60 PRINT "AND PASSED LOGICAL TEST"
]70 END
]RUN
AND PASSED LOGICAL TEST
```

In the preceding example, line 30 first tested the value of A. Since A was set equal to 2 in line 10, the first *expression* was evaluated as true. The value of B was then tested. It too evaluated as true. Using the logical AND table, since *expression1* and *expression2* evaluated to true, the whole AND *expression* evaluated as true. The program will then execute the THEN portion of the statement and will branch to line 60. At line 60, the message AND PASSED LOGICAL TEST was displayed.

Example 2

```
PRINT (3 = 1 + 2) AND (-5)
1
```

In this example, 3 is compared to 1 + 2, so the first *expression* evaluates as true. The second *expression* (-5) is non-zero, so it is also evaluated as true. According to the AND truth table, if both *expressions* evaluate as true, then the whole *expression* is true. SmartBASIC represent true as 1, so a 1 is printed.

ASC

The ASC function returns the ASCII code for the first character in its *argument.*

Configuration

ASC(*argument*)

argument can be any string variable or constant.

Example

```
]A$ = "A"

]PRINT ASC(A$), ASC("DEF")
65                  68
```

. the preceding example, the character in the string A$ was A. A's ASCII equivalent is 65. In the second string, the first character D will be used as the argument. A value of 68 is returned for the ASCII value of D.

ATN

The ATN function is a trigonometric function that returns the arctangent of its *argument.*

Configuration

ATN(*argument*)

The *argument* can be a numeric expression or numeric constant. The value returned will be the primary angle $(-\frac{\pi}{2} < \text{angle} < \frac{\pi}{2})$.

Example

```
]10 PI = ATN(1) * 4
]20 PRINT PI, ATN(TAN(.2))
]RUN
3.14159265                    .2
```

In the preceding example, the arctangent of 1 returns the value $\pi/4$. Multiplying this value by 4 returns the value indicated.

In the second part of the PRINT statement, the argument .2 is returned. Since the ATN formula is the inverse of the TAN function, th value returned was the original argument.

CALL

The CALL statement is used to execute a machine language subroutine.

Configuration

CALL *expression*

expression evaluates to an integer between -65535 and +65535. The *expression* is the location of the machine language subroutine.

In SmartBASIC, there are two values which will execute the same machine language subroutine. There is the positive address and the negative address. The conversion is as follows:

positive address - 65536 = negative address

Example

```
CALL 20996
```

The preceding CALL executes a machine language subroutine at the given location. This CALL is identical to the CATALOG command.

CHR$

The CHR$ function returns the ASCII character for the value given in the *argument*.

Configuration

CHR$(*argument*)

argument is a real number or an integer between 0 and 255. If the *argument* is a real number, its decimal portion will be truncated.

Example

```
]10 X$ = CHR$(80)
]20 PRINT CHR$(65), X$
]RUN
A                      P
```

The ASCII code for A is 65 and the code for P is 80.

CLEAR

CLEAR initializes all variables, arrays, and strings to zero. CLEAR also initializes all DATA pointers, FOR,NEXT counters, subroutine pointers, etc.

Configuration

CLEAR

CLEAR can be used anywhere in a program, but should not be used in a subroutine or FOR,NEXT loop.

Example

```
]10 A = 10
]20 PRINT A
]30 CLEAR
]40 PRINT A
]RUN
10
0
```

Line 30 sets variable A from 10 to 0.

COLOR

The COLOR statement defines the next color to be displayed by the graphics statements PLOT, HLIN,AT, and VLIN,AT.

Configuration

COLOR = *expression*

The *expression* is an integer from 0 to 15. The computer can display a total of 16 different colors. The colors and their associated numbers are shown below.

0	Black	8	Yellow
1	Magenta	9	Medium Red
2	Dark Blue	10	Grey
3	Dark Red	11	Pink
4	Dark Green	12	Light Green
5	Grey	13	Light Yellow
6	Medium Blue	14	Cyan
7	Light Blue	15	White

Example

```
]10 GR
]20 COLOR = 6
]30 PLOT 0,0
]40 END
```

The preceding program will place a blue square in the upper left hand corner of the screen.

CONT

CONT resumes program execution at the next instruction.

Configuration

```
CONT
```

This command is generally executed following a STOP, END, or CONTROL-C.

Example

```
]10 FOR I = 1 TO 5
]20 PRINT I, I ^ 2
]30 IF I = 3 THEN STOP
]40 NEXT I
]RUN
1               1
2               4
3               9

BREAK IN 30
]CONT
4               16
5               25
```

In the preceding example, program execution stopped in line 30 when I = 3. Typing in CONT continued program execution.

COS

The COS function is a trigonometric function that returns the cosine of its *argument.*

Configuration

COS(*argument*)

The *argument* is a numeric expression or numeric constant in radians.

Example

```
]PRINT COS(3.141592653)
-1
```

In the preceding example, the cosine of π is returned.

DATA

The DATA statement contains a list of data items. These data items are read into the variables specified by the READ statement.

Configuration

DATA *item* [,*item*...]

item can either be a real number, integer, or string. The data items must be in the same order as they are intended to be read by corresponding READ statements.

If a comma or colon is to be included in the string, the item should be enclosed in quotes. The following characters cannot be placed as data in a DATA statement.

RETURN
HOME
"
any arrow
TAB

CONTROL-H
CONTROL-M
Backspace
CONTROL-X

The preceding characters may be used in a program by executing the CHR$ function.

The DATA statement can be located anywhere in a program. It does not have to precede the READ statement.

Example

```
]10 DATA "SMITH, JOE", JOHN BROWN
]20 READ N1$, N2$
]30 PRINT N1$, N2$
]RUN
SMITH, JOE           JOHN BROWN
```

The READ interpreted the first string as SMITH, JOE because it was enclosed in quotes. The second string is read as JOHN BROWN.

DEF FN

The DEF FN statement allows the user to define a function. This function can then be used in the same manner as any built-in function.

Configuration

DEF FN *name* (*variable*) = *expression*

name is the name of the function. Like variable names, only the first two characters are significant. The *variable* can be any real numeric variable name. The *expression* can be a numeric constant or a numeric equation.

Example

```
]10 PI = ATN(1) *4
]20 DEF FNAR(X) = PI * X ^ 2
]30 FOR RAD = 1 TO 3
]40 PRINT RAD, FNAR(RAD)
]50 NEXT RAD
```

```
]60 END
]RUN
1               3.14159265
2               12.5663706
3               28.2743339
```

In line 10, PI is calculated so it can be used in the function definition. In line 20, the function for the area of a circle is defined. Line 40 then uses the function by passing the value of the radius to the function. The value of the area of the circle is then returned and printed by the PRINT statement.

DEL

DEL deletes the lines given in the argument.

Configuration

DEL *a* [,*b*]

a and *b* are integers greater than or equal to 0. *b* must be greater than *a*. If *a* is not an existing line number in the program, the next highest line number will be used. If *b* is not an existing line number in the program, the next lowest line number will be used.

The DEL can also be used as a program statement in SmartBASIC. If the DEL is used as a program statement, the specified lines will be deleted. However, program execution will halt after the statement has been executed. The CONT command will not resume program execution.

An example of the use of the DEL command can be found on the following page.

Example

```
]10 TEXT
]20 HTAB 12
]30 VTAB 3
]40 PRINT "HELLO"
]50 END
]DEL 30,50
]LIST
10 TEXT
20 HTAB 12
```

DIM

The DIM statement is used to allocate memory space for strings, arrays, or matrices.

Configuration

a (*i*[,*j*]...) [,*b* (*i*[,*j*]...)]
DIM *a%* (*i*[,*j*]...) [,*b%* (*i*[,*j*]...)]
a$* (*i*[,*j*]...) [,*b$ (*i*[,*j*]...)]

a and *b* are the variables to be dimensioned. *i* and *j* are integers.

All arrays, strings, and matrices are predefined with subscripts of 10. Above 10, the value in a DIM statement corresponds to the largest subscript that can be used in that variable. However, there is always a zero subscript. As a result, to save 100 values in a single dimension array, the correct DIM statement would be DIM A(99).

The maximum size of strings and arrays depends on the amount of available memory at the time the DIM statement was executed.

If the DIM statement exceeds the amount of available memory, the following error will occur:

?OUT OF MEMORY ERROR IN *line*

where *line* is the line of the DIM statement.

Example

```
DIM A$(10,5), C%(2,20)
```

In the preceding example, 66 string spaces are allocated for A$, and 63 integer variables are defined for C%.

DRAW

The DRAW statement plots a shape on the high-resolution graphics page.

Configuration

DRAW *shapeno* [AT *X*, *Y*]

shapeno is an integer between 0 and 255. *X* and *Y* are integers for the position of the shape. *X* must lie between 0 and 255. *Y* must lie between 0 and 191.

If AT *X*, *Y* is not given, the shape will be plotted at the last X, Y position designated.

The color, rotation, and size of the shape must have been previously defined.

Example

```
]10 HGR2
]20 HCOLOR = 9
]30 SCALE = 20
]40 FOR I = 0 TO 62 STEP 2
]50 ROT = I
]60 DRAW 1 AT 128,96
]70 NEXT I
```

In the preceding example, line 10 initialized the screen for high resolution graphics. Lines 20 and 30 then define the color and size of the shape drawn at line 60. The FOR,NEXT loop, lines 40-70, vary the rotation and redraw the shape.

END

The END statement is used to stop program execution.

Configuration

```
END
```

The END statement is optional in SmartBASIC. If it is not used, the program will stop execution at the highest line number.

Example

```
999 END
```

EXP

The EXP function returns the value of e raised to the power of the *argument* (e = 2.71828183).

Configuration

EXP (*argument*)

he argument must be a numeric constant or numeric expression.

Example

```
PRINT EXP(5)
148.413159
```

In the preceding example, e^5 was returned.

FOR,NEXT

The FOR,NEXT statements are used to execute a sequence of statements a set number of times.

Configuration

```
FOR variable = a to b [STEP c]
  .
  .
  .
NEXT [variable] [,variable ...]
```

variable is a real variable in SmartBASIC. The *variable* is used as a counter. *a*, *b* and *c* are numeric expressions or constants. *a* is the initial value of the counter and *b* is the final value. The counter is incremented or decremented depending on the sign of *c*. If *c* is not given, it will be assumed as 1.

The program lines following the FOR statement will be executed until the NEXT statement is encountered. At this point, the counter is incremented (assuming positive STEP value) by the STEP value.

The value for the counter is then compared with its final value *b*. As long as the counter's value does not exceed the final value, the program will branch back to the statement following the FOR statement. This entire process will then be repeated.

When the counter's value exceeds the specified final value (*b*), the statement following the NEXT statement will be executed. This will exit the FOR,NEXT loop.

One FOR,NEXT loop may be placed within another FOR,NEXT loop. This is known as **nesting** or **nested loops**. When FOR,NEXT loops are nested, each FOR,NEXT loop must use a different variable name for the counter. Also, the NEXT statement for the inside loop must appear before the NEXT statement for the outside loop. However, if both loops end at the same point, a single NEXT statement may be used to end these. Be certain that the variable for the inside loop appears before the variable for the outside loop. A NEXT statement such as the following:

```
NEXT J,I
```

would be interpreted as follows:

```
NEXT J
NEXT I
```

Example

```
]10 FOR X = 1 TO -2 STEP -1
]20 PRINT X
]30 NEXT X
]40 END
]RUN
1
0
-1
-2
```

In the preceding example, the STEP value is -1 so the counter is decremented until its value is -2.

FRE

The FRE function returns the number of free bytes in memory.

Configuration

FRE (*argument*)

argument can be any legal expression. It makes no difference what the *argument* is.

When FRE is used, a housekeeping will be performed before the function returns the number of free bytes. Housekeeping is a process where BASIC gathers all useful data by freeing any memory which was once used for strings, but which is currently unused. Memory for strings becomes unused when the string's length changes.

Example

```
]10 A = FRE (0)
]20 PRINT "NUMBER OF FREE BYTES IS";A
]30 END
]RUN
NUMBER OF FREE BYTES IS 26004
```

GET

The GET statement inputs a single character from the keyboard. The character is not displayed on the screen.

Configuration

GET *variable*

variable can be any legal SmartBASIC variable.

Although *variable* can be any variable, it is to the user's advantage to use a string variable and convert it to a numeric variable with the VAL function. If a numeric variable was used with GET, any non-numeric character entered will cause a syntax error and halt program execution.

Example

```
]10 GET A$
]20 PRINT A$;
]30 IF A$ = CHR$(3) THEN END
]40 GOTO 10
```

The preceding example demonstrates the use of the GET statement. This program accepts a single character from the keyboard and displays the character on the screen. The GOTO statement at line 40 causes the GET statement to be repeated. The conditional statement at line 30 causes the program to end when the CONTROL-C combination is entered.

GOSUB, RETURN

The GOSUB, RETURN statements are used to branch to a subroutine and then return from it.

Configuration

```
GOSUB line
  .
  .
  .
RETURN
```

line is the first line of a subroutine. A subroutine is called by the GOSUB statement. When the RETURN statement is encountered within that subroutine, program control will branch back to the statement following the GOSUB statement just executed.

Subroutines may appear at any point within the program. However, it is good programming practice to group all subroutines at the end of the program, and to include an END statement to separate the main program from the subroutines.

Example

```
]10 X = 0
]20 FOR I = 1 TO 3
]30 X = X+1
]40 GOSUB 70
]50 NEXT I
]60 END
]70 PRINT X,
]80 Y = X*X
]90 PRINT Y
]100 RETURN
]RUN
1          1
2          4
3          9
```

When the program reaches line 40, the GOSUB will be executed. The program branches to line 70. When the RETURN in line 100 is reached, program execution jumps back to line 50. This process continues until the FOR counter reaches 3.

GOTO

The GOTO statement branches program control to another program line.

Configuration

GOTO *line*

line is the line number of the statement to be branched to.

Example

```
]10 PRINT "FIRST"
]20 GOTO 40
]30 PRINT "MIDDLE"
]40 PRINT "LAST"
]50 END
]RUN
FIRST
LAST
```

GR

The GR statement sets and clears the low resolution screen mode (40x40 with 4 lines of text at the bottom of the screen).

Configuration

GR

This statement should be executed before the graphics statements PLOT, HLIN, AT, and VLIN,AT are used.

When the GR statement is executed, the color is automatically set to 0 (BLACK).

Example

```
]10 GR
]20 COLOR = 15
]30 PLOT 19, 23
]40 END
```

The preceding example should put a white square on the screen. If line 10 was omitted, the PLOT command in line 30 would have a null effect, because the low-resolution mode was not set.

If you wish to return to the normal mode, you can do so by executing the TEXT statement.

HCOLOR

The HCOLOR statement defines the next color to be displayed by the graphics statements, HPLOT and DRAW. HCOLOR is used in the high resolution graphics mode.

Configuration

HCOLOR = *number*

number is a numeric expression or numeric constant that evaluates to a real number or integer between 0 and 15. Values outside this range will produce an error. The colors and their associated numbers are shown below.

0 - Black
1 - Green
2 - Violet
3 - White
4 - Black
5 - Orange
6 - Blue
7 - White

8 - Brown
9 - Dark Blue
10 - Grey
11 - Pink
12 - Dark Green
13 - Yellow
14 - Aqua
15 - Magenta

HPLOT and DRAW will all output lines in the color indicated by HCOLOR until a subsequent HCOLOR statement is executed.

HGR

The HGR statement sets and clears the high-resolution graphics mode (256 x 160), with 4 lines of text at the bottom of the screen.

Configuration

```
HGR
```

Output to the four lines of text may be accomplished by using the standard PRINT command.

Example

```
]HGR
```

When the preceding example is executed, the high-resolution graphics mode will be set and the screen will be cleared to black. There will be four lines of text at the bottom. The high resolution color will automatically be set to 0 (black).

HGR2

HGR2 sets the screen to the high resolution graphics mode without the text lines at the bottom of the display (256 x 192).

Configuration

```
HGR2
```

Except that no text display is supported, HGR2 operates identically to HGR.

Example

```
]10 HGR2
]20 HCOLOR = 9
]30 HPLOT 128,0 TO 211,144
]40 HPLOT TO 45,144 TO 128,0
]50 HPLOT 45,48 TO 211,48
]60 HPLOT TO 128,191 TO 45,48
```

HIMEM

The HIMEM statement defines the address of the highest memory location available to a BASIC program.

Configuration

HIMEM: *number*

number is a numeric constant or numeric expression. The value of *number* should indicate the highest available memory address. This value must lie between -65535 and 65535.

If the HIMEM: is set lower than LOMEM or set so low that there is not enough room for the program to run, an out of memory error will occur.

The value of HIMEM is reset by the commands NEW, LOMEN, CLEAR and RUN. Therefore, HIMEN should be used in programs.

The HIMEM statement is generally used to reserve memory for a machine language subroutine called by the BASIC program. The HIMEM statement keeps BASIC variable and array storage separate from the machine language subroutine.

Example

```
HIMEM: 33024
```

The preceding example sets high memory to memory address 33024. SmartBASIC will now only use memory up to this address.

HLIN

HLIN is used in the low resolution graphics mode to draw a horizontal line on the screen.

Configuration

HLIN *column 1, column 2* AT *row*

column 1, *column 2*, and *row* can be either numeric constants or numeric expressions. *column 1* and *column 2* must lie in the range of 0 to 39. Also, the value of *column 1* must be less than or equal to *column 2. Row* must lie in the range of 0 to 39.

If an incorrect value is used for *column 1*, *column 2*, or *row*, the following error message will be displayed:

?ILLEGAL QUANTITY ERROR

If HLIN is executed in the text mode, a null result is achieved.

Example

```
]10 GR
]20 COLOR = 3
]30 HLIN 0, 39 AT 20
]40 END
```

The preceding example will draw a dark red line across the screen at row 20.

HOME

The HOME statement clears the screen and places the cursor in the upper left hand corner of the text screen.

Configuration

HOME

If HOME is executed while a graphics plus text format is displayed, only the text window is cleared.

Example

```
]HOME
```

HPLOT

The HPLOT statement can be used to place a dot or draw a line on the high resolution graphics screen. The color of the dot must have been previously defined by the HCOLOR statement.

Configuration

HPLOT *column 1, row 1* [TO *column 2, row 2* ...]
HPLOT TO *column, row*

column, *row*, *column 1*, *column 2*, *row 1*, and *row 2* are numeric constants or numeric expressions. *column*, *column 1*, and *column 2* must lie between 0 and 255. The *row*, *row 1*, and *row 2* must lie between 0 and 191.

If the HPLOT is used as shown in the first configuration without the optional (TO *column 2*, row 2), a dot will be plotted. The optional TO will connect the two dots. If the *column 1* and *row 1* preceding the TO are omitted, the line will be drawn from the previous point plotted to the point indicated by *column 2, row 2.*

Example

```
]10 HGR
]20 HCOLOR = 3
]30 HPLOT 0,0
]40 HPLOT TO 0,50 TO 50,50
]50 HPLOT TO 50,0 TO 0,0
```

The preceding example will draw a square in the upper left hand corner of the screen.

HTAB

The HTAB statement positions the cursor at the location specified by its *argument.*

Configuration

HTAB *argument*

argument is a numeric constant or numeric expression. The *argument* must be between 0 and 255.

The cursor will be moved to the position specified by the *argument.* HTAB moves the cursor without erasing any displayed characters. If *argument* is greater than the length of a physical line (31 characters), the subsequent lines will be used until the correct position is attained.

Example

```
]10  PRINT "1234567890"
]20  HTAB 3 : PRINT 3;
]30  HTAB 5 : PRINT 5;
]40  HTAB 9 : PRINT 9
]RUN
1234567890
  3 5   9
```

In the preceding example, line 20 places the cursor at position 3 and displays a 3. In line 30, the cursor is moved to position 5. The PRINT statement displays a 5. In line 40, HTAB moves the cursor to position 9, and the PRINT statement displays a 9.

IF,THEN

The IF,THEN statement sets up a condition which will influence the program flow.

Configuration

IF *expression* THEN *statement* [:*statement*...]

expression is a conditional expression. *statement* can be any BASIC statement.

If the *expression* is evaluated as true, the THEN portion of the statement will be executed.

In SmartBASIC, if the *expression* evaluates as true, the *statement* following THEN will be executed. If the *expression* evaluates as false, the statement in the next program line will be executed.

For example, consider the following statement.

```
IF X=15 THEN PRINT "TRUE":PRINT X
```

If the variable X equals 15, the following would be displayed:

```
TRUE
15
```

If X does not equal 15, TRUE would not be displayed, nor would the value of X.

If only a number is placed after THEN, a GOTO that line number is executed when the statement is true.

```
10  IF A = 15 THEN 90
```

or

```
10  IF A = 15 THEN GOTO 90
```

INT

The INT function returns the integer value of the specified *argument.*

Configuration

INT (*argument*)

argument is a numeric constant or numeric expression.

The value returned will always be less than or equal to the original value.

Example

```
PRINT INT (1.7), INT (-1.7)
1          -2
```

INVERSE

The INVERSE statement turns on the INVERSE (reverse) video. Following the execution of the INVERSE statement, any characters displayed by the computer will be in inverse (i.e. characters will be displayed as black characters on a white background.)

Configuration

```
INVERSE
```

The INVERSE mode works by altering the standard ASCII code only in the display. Therefore, the inverse video characters can only be saved on the Digital Data Packs if the ASCII codes for them are written to the drive. The printer cannot support inverse video.

The INVERSE statement can be turned off by the NORMAL statement.

Example

```
] INVERSE
] PRINT "++"
++
```

Note that the PRINT statement as well as the output will appear in reverse video on the display.

INPUT

The INPUT statement accepts data entry from the keyboard or another input device while the program is being executed.

Configuration

INPUT [*"message"*;] *variable* [,*variable*]

message is a string used as a prompt. *variable* can be any valid BASIC variable.

When an INPUT statement is executed, program execution will stop temporarily. If a prompt was included, the prompt will be displayed. A question mark will be displayed if there is no prompt.

After the INPUT statement has been executed, the user may enter the desired data at the keyboard. That data is assigned to the *variable(s)* listed in the INPUT statement. The number of data items entered must equal the number of *variables* listed. Also, the type of data entered must agree with the type specified in *variable*. The data items must be delimited by commas when input. Because the comma is used as delimiter, if a comma is to be entered in a string, quotes must be placed around the string data.

Example

```
]10 INPUT "ENTER A NUMBER "; A
]20 PRINT "THE NUMBER IS "; A
]30 END
]RUN
ENTER A NUMBER 4.5←———user's response
THE NUMBER IS 4.5
```

LEFT$

The LEFT$ function returns the number of characters specified in the second expression of the argument to the leftmost of the string specified in the first part of the argument.

Configuration

LEFT$ (*a$*,*x*)

a$ is a string constant searched by the function. *x* is the number of characters to be returned.

Example

```
]10  A$ = "ABCDEFG"
]20  PRINT LEFT$ (A$, 3)
]30  END
]RUN
ABC
```

The preceding LEFT$ function returned the 3 leftmost characters in A$.

If the value of the numeric argument exceeds the length of the string argument, the entire string value will be returned. If the value of the numeric argument is less than one or greater than 255, the following error message will be displayed:

```
?Illegal Quantity Error
```

LEN

The LEN function returns the number of characters in a string.

Configuration

```
LEN (a$)
```

a$ is a string constant.

Example

```
PRINT LEN ("ADAM")
4
```

LET

The LET statement is an optional assignment statement. An assignment statement determines the value of an expression and then assigns that result to the variable named in the assignment statement.

Configuration

LET *variable* = *expression*

variable must be of the same data type as the expression. For example, if *variable* is a string, *expression* must also be a string. If *variable* is an integer or real number, then *expression* must also be numeric.

Example

```
]10 LET C = 1+A
]20 L = C*2
```

LIST

The LIST command is used to list the program stored in memory on the video display or other device.

Configuration

LIST *a* [$\{\overset{,}{-}\}$[*b*]]

or

LIST [[*a*]$\{\overset{,}{-}\}$]*b*

a and *b* are integers greater than or equal to 0. If *a* is greater than *b*, no lines will be listed.

If *a* is not a line number in the program, the next highest line number will be used. If *b* is not a line number in the program, the next lowest line number will be used.

LIST can be frozen by CONTROL-S. Pressing any other key will resume LIST. The listing may be stopped by pressing CONTROL-C.

Example

```
]10 PRINT "My"
]20 PRINT "name"
]30 PRINT "is"
]40 PRINT "Adam"
]LIST 10
  10 PRINT "My"
]LIST 20,40
  20 PRINT "name"
  30 PRINT "is"
  40 PRINT "Adam"
```

LOAD

The LOAD command is used to load a program from a storage device to the computer.

Configuration

LOAD *filename* [,D *drive*]*

filename is the name of the program. *drive* is the number of the Digital Data Drive that the file is in.

LOAD need only be entered with the program name, and the RETURN key pressed. If the indicated filename is not present on the specified Data Drive, the File Not Found error will occur.

Example

```
]LOAD PROGRAM
```

* WARNING: If an attempt is made to access a drive which is not present in your system, the system I/O will be disabled. Reset the computer to recover.

LOG

The LOG function returns the natural log of the argument.

Configuration

```
LOG (argument)
```

argument is a numeric constant or numeric expression greater than 0.

The natural log is undefined for negative numbers.

Example

```
]PRINT LOG(25)
3.21887582
```

LOMEM

The LOMEM statement defines the address of the lowest memory location available for BASIC.

Configuration

```
LOMEM: number
```

number is a numeric constant or numeric expression. The value of *number* should be the address of the lowest memory available. This value must lie between -65535 to 65535.

LOMEM cannot be set lower than 27294, nor can it be set lower than its current value. LOMEM can only be increased.

LOMEM will be reset by the NEW or DEL commands or by adding or changing a line.

Example

```
LOMEN:28000
```

MID$

The MID$ function returns the portion of a string specified by its argument.

Configuration

MID$(*a$*, *b*[,*c*])

a$ is a string constant. *b* and *c* are numeric constants or numeric expressions with a value between 0 and 255. *b* is the first character in *a$* being returned. *c* is the number of characters in *a$* being returned. If *c* is not included, all characters to the right of the position given in *b* will be returned.

Example

```
]10 N$ = "COMPUTER"
]20 PRINT MID$(N$, 4, 3)
]30 END
]RUN
PUT
```

In the preceding example, the fourth position in the string N$ is the starting position. The 3 indicates 3 characters.

If the value of the numeric argument exceeds the length of the string argument, the entire string value will be returned. If the value of the numeric argument is less than one or greater than 255, the following error message will be displayed.

```
?Illegal Quantity Error
```

NEW

The NEW command deletes the program in memory and clears all variables.

Configuration

```
NEW
```

The NEW command is generally used to free memory space before a new program is entered.

Example

```
]10 TEXT
]20 END
]LIST
  10 TEXT
  20 END
]NEW
]LIST
```

NORMAL

The NORMAL statement turns off the INVERSE mode.

Configuration

```
NORMAL
```

The NORMAL statement sets the video output mode to white characters on a black background.

Example

```
]NORMAL
```

NOT

The NOT function logically compliments the value given in the *argument.*

Configuration

NOT *argument*

argument is a numeric constant or numeric expression. If the *argument* evaluates to true (non-zero), false (zero) will be returned. If the argument evaluates to false (zero), true (one) will be returned.

```
NOT 1 = 0
NOT 0 = 1
```

Example

```
]10  A = 2
]20  IF NOT (A = 1) THEN PRINT "A DOES NOT EQUAL ONE"
]30  END
]RUN
A DOES NOT EQUAL ONE
```

NOTRACE

The NOTRACE command turns off the TRACE command.

Configuration

NOTRACE

The NOTRACE command may be used as a program statement.

Example

```
]NOTRACE
```

ON

The ON statement is used in conjunction with GOTO and GOSUB. The statements are used to branch program control to one of several program lines depending on the value appearing after ON.

Configuration

ON *exp* GOTO *line* [,*line* ...]
ON *exp* GOSUB *line* [,*line* ...]

exp can be any numeric constant or numeric expression. *line* is the line number the program is to branch to.

The value of *exp* controls which *line* is to be branched to. For instance, if *exp* evaluates to 1, program control will branch to the line number given in the first *line*. If *exp* evaluates to 2, program control will branch to the second *line*, etc...

If the ON,GOSUB statement is being used, the line number specified in *line* can be that of a subroutine. In other words, a RETURN statement eventually can be executed to return program control.

If *exp* evaluates to zero or to a number greater than the number of *lines* specified after GOTO or GOSUB, the program will continue with the next executable statement.

Example

```
]10 INPUT "ENTER AN INTEGER FROM 1 TO 4 ";I
]20 ON I GOTO 60,80,100,120
]30 PRINT
]40 INPUT "PLEASE ENTER AN INTEGER FROM 1 TO 4 ";I
]50 GOTO 20
]60 PRINT "YOU ENTERED A ONE"
]70 GOTO 130
]80 PRINT "YOU ENTERED A TWO"
]90 GOTO 130
]100 PRINT "YOU ENTERED A THREE"
]110 GOTO 130
]120 PRINT "YOU ENTERED A FOUR"
]130 END
```

In the preceding example, line 10 prompts the user to enter a number between one and four. In line 20, an ON,GOTO will branch control to a different line depending on the value of I. If I is one, program execution will branch to 60. If I is two, program execution will branch to 80, etc. If zero or a number greater than four was entered, program execution will continue to line 30.

ONERR GOTO

The ONERR statement allows errors to be trapped. The statement then transfers program control to an error handling routine at the indicated line number.

Configuration

ONERR GOTO *line*

line is the first line of the error handling routine. ONERR GOTO should be executed before the error has occurred.

When SmartBASIC executes a program, it executes the program line by line. If an error occurs during program execution, SmartBASIC will check to see if an ONERR GOTO statement has been executed. If no ONERR GOTO statement had been executed, SmartBASIC will halt program execution and display the error. Otherwise, the program will branch to the *line* indicated in the ONERR GOTO statement.

The RESUME command can be used to return the program to the beginning of the statement where the error occurred.

OR

OR is a logical operator. This reserved word is generally used in conjunction with the IF,THEN statement.

Configuration

expression 1 OR *expression 2*

expression 1 and *expression 2* are Boolean expressions. If the *expression* is numeric (non-zero), it will be evaluated to true. A zero is treated as false. A truth table for OR is illustrated below.

X	Y	X OR Y
true	true	true
true	false	true
false	true	true
false	false	false

In SmartBASIC, a true is represented by a 1 and false by a 0.

Example

```
]10 A = 3
]20 B = 5
]30 IF (B < A) OR (B = 5) THEN 50
]40 END
]50 PRINT "EITHER B IS LESS THAN A"
]60 PRINT "OR B IS EQUAL TO 5"
]70 END
]RUN
EITHER B IS LESS THAN A
OR B IS EQUAL TO 5
```

In the preceding example, B is not less than A, but B is equal to 5. Therefore, the whole OR expression is true, and the program branches to line 50.

PDL

The PDL function returns the value corresponding to a specific game controller manipulation.

Configuration

PDL (*argument*)

argument is a numeric constant or numeric expression. The value of *argument* must lie between 0 and 15. The value of *argument* corresponds to the following manipulations.

controller number	argument	value returned
#2	0	Pushing the joystick up causes the variable that represents vertical position to decrease; pushing down increases the value of the variable. The variable will always lie in the range 0-255.
#1	1	
#2	2	Pushing the joystick to the right causes the variable that represents horizontal position to increase; pushing down decreases the value of the variable. The variable will always lie in the range 0-255.
#1	3	

#2	4	Returns value corresponding to the current joystick position.
		UP = 1 DOWN-RIGHT = 6
		RIGHT = 2 DOWNLEFT = 12
		DOWN =4 UP-LEFT = 9
		LEFT = 8 CENTER = 0
		UP-RIGHT = 3

#1	5	
#2	6	Left button Pressed = 1 Released = 0
#1	7	
#2	8	Right button Pressed = 1 Released = 0
#1	9	
#2	10	ASCII coded value of key pressed on numeric keypad; no key pressed = 0
#1	11	
#2	12	Numeric value of key pressed on keypad. "*" = 10 "#" = 11 no key pressed = 15
#1	13	
	14	Not yet implemented
	15	

Example

```
]10 VTAB 1
]20 FOR I = 0 TO 15
]30 PRINT I;TAB(4);PDL(I);
]40 NEXT I:GOTO 10
```

The preceding program displays the 16 different values returned by the PDL function. By manupulating either hand controller, the values returned by PDL will change.

PEEK

The PEEK function returns the contents of the memory address given in the *argument*.

Configuration

PEEK (*argument*)

argument is a numeric constant or numeric expression between 65535 and 65535.

The decimal integer returned by the function will lie between 0 and 255.

Example

```
]PRINT PEEK(64000)
203
```

The preceding example returns the contents of location 64000.

PLOT

The PLOT statement plots a dot on the low resolution graphics screen. The color of the dot must have been previously defined by the COLOR statement.

Configuration

PLOT *column, row*

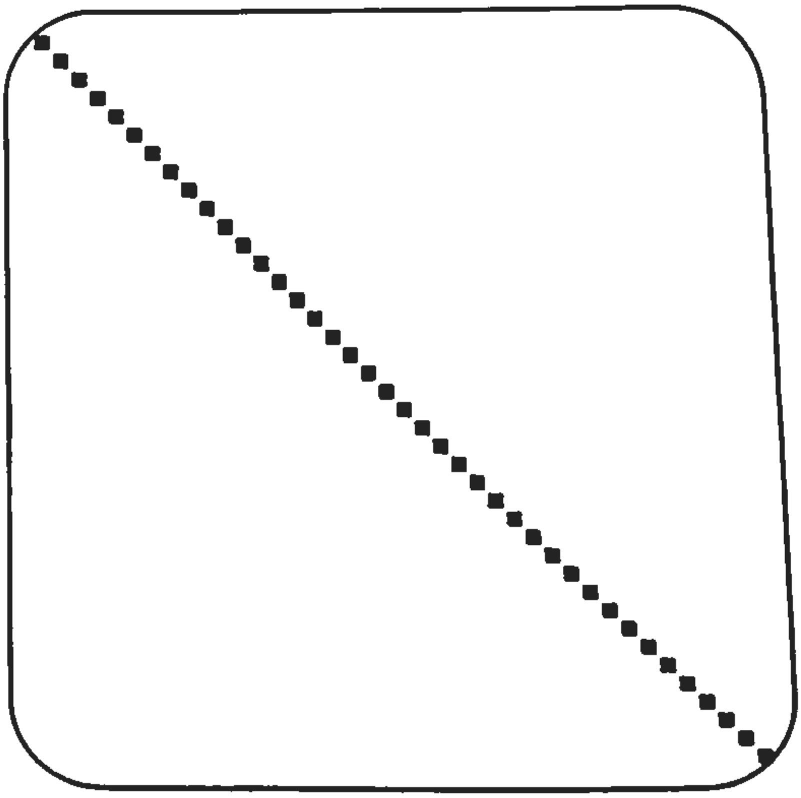
column and *row* must be numeric constants or numeric expressions. Both *column* and *row* must lie between 0 and 39.

PLOT occurs at the position specified. For example PLOT 3, 5 would place a dot at row five and column three. The origin (0,0) is located in the upper left hand corner of the screen. If a PLOT statement is executed when either the text mode or the high resolution mode is active, the screen is not affected.

Example

```
]10 GR
]20 COLOR = 3
]30 FOR I = 0 TO 39
]40 PLOT I,I
]50 NEXT I
]60 END
```

The preceding example will draw a diagonal line across the screen.



POKE

The POKE statement stores one byte of information in the memory location specified.

Configuration

POKE *address, value*

address and *value* are numeric constants or numeric expressions. *address* lies between -65535 and 65535. *value* must lie between 0 and 255.

The POKE statement places the indicated *value* at the specified memory *address*. A POKE has no effect if the *address* is in ROM. If a POKE is not used carefully, it can disrupt the ADAM's execution.

Example

```
]10 PRINT PEEK(32000)
]20 POKE 32000,37
]30 PRINT PEEK(32000)
]40 POKE 32000,158
]50 PRINT PEEK(32000)
]60 END
```

In the preceding example, line 10 first displays the current contents of memory location 32000. The value 37 is then POKE'd into memory on line 20. Line 30 displays the value at memory location 32000. The value 158 is then POKE'd into memory and displayed in lines 40 and 50.

POP

The POP statement causes a program to ignore the GOSUB or ON,GOSUB statement that was executed last.

Configuration

POP

In effect, a GOSUB or ON,GOSUB statement is converted to a GOTO or ON,GOTO statement when POP is executed. The program "forgets" that it is in a subroutine. As a result, when a POP statement is executed, the next RETURN statement branches the program control to the line after the GOSUB statement before the previous GOSUB statement. In other words, the program "forgets" where the subroutine was called from, so it returns to a previous GOSUB statement.

A POP statement is used, in general, to exit a subroutine.

Example

```
]10 X = 5
]20 Y = 10
]30 GOSUB 100
]40 END
]100 PRINT X
]110 IF X > 0 THEN POP:GOTO 130
]120 RETURN
]130 PRINT Y
]140 END
]RUN
5
10
```

The previous example contains a program that uses a POP statement to exit a subroutine. At line 10, X is assigned the value 5. At line 20, Y is assigned the value 10. At line 30, the subroutine at line 100 is called.

At line 100, the value of X is displayed. Line 110 is an IF,THEN statement that tests the condition X > 0. Since the value of X is greater than zero, the condition is true. As a result, the POP statement is executed, and the program control branches to line 130. At line 130, the value of Y is displayed.

Because the POP statement was executed, the program no longer remembers the subroutine. If a RETURN statement were to be executed in line 140, the program would not return to the statement following line 30. In fact, a RETURN statement in line 140 would cause an error, since the program would not know where to branch.

If a GOSUB had been executed that transfered program control to line 10, and if a RETURN statement was present in line 140, the RETURN in 140 would transfer program control back to the statement following the GOSUB that called line 10.

A POP statement can also be used to make the program ignore the previous FOR statement. When a POP statement is executed within a FOR,NEXT loop, the loop will not be repeated. However, an error occurs if a NEXT statement is executed for that loop.

POS

The POS function returns the current horizontal position of the cursor.

Configuration

POS (*argument*)

argument can be any legal constant or expression.

The number returned will be an integer from 0 to 39. The leftmost position is 0.

Example

```
]HTAB 9 : PRINT POS(0)
            8
]PRINT TAB(9); POS(0)
            8
]PRINT SPC(9); POS(0)
             9
```

In the previous example, the HTAB and TAB function count the leftmost position as 1. The SPC and POS function treat the leftmost position as 0.

PRINT

PRINT is used to display information to the screen or to another output device.

Configuration

PRINT [*expression*] [;...[*expression*]...]

expression can be any valid numeric or string constant or expression. *expression* can include string and numeric variables, as well as string and numeric constants. Each variable name or constant must be separated by either a comma or a semicolon. When a comma separates the items in a PRINT statement, the display is divided into two display positions in SmartBASIC. These begin in columns 1 and 17.

A PRINT statement can end with a comma, semicolon, or with no punctuation at all. A PRINT statement that ends with a semicolon causes any subsequent PRINT statement output to appear at the next position on the same row of output.

When a PRINT statement ends with a comma, the next PRINT statement output will occur at the next PRINT display position on the same row of output.

When a PRINT statement includes no ending punctuation, the next line of output will automatically occur on the next display line. A PRINT statement always clears the screen from the last displayed character to the end of the line.

PR#

PR# specifies the peripheral which will be providing subsequent output for the ADAM.

Configuration

PR# *argument*

argument is a numeric constant or numeric expression which specifies the peripheral. The value of the *argument* must lie between 0 and 7.

argument	peripheral
0	screen
1	printer
2-7	not yet supported

WARNING: If a PR#8 is executed, the system I/O will be disabled.

READ

A READ statement is used to assign values to variables. The values are taken individually from DATA statements in the order they appear in the program.

Configuration

$$\text{READ}\ \begin{matrix} a \\ a\$ \end{matrix} \left[\begin{matrix} ,b \\ ,b\$ \end{matrix}\ \ldots\right]$$

Data items are assigned to variables in the order in which they appear in the program unless a RESTORE statement has been executed.

The type of variable in the READ statement must correspond to the type of data in the corresponding DATA statement. A numeric variable can only be assigned a numeric value. However, a string variable can accept any type of character or none at all.

A program must include at least as many data items as the number of variables in its READ statements unless a RESTORE statement is executed.

Example

```
]20 READ X,X$
]30 PRINT X$,X
]40 END
]50 DATA 12, JONES
]RUN
JONES     12
```

The preceding example contains a program that has a READ statement. At line 20, the variables X and X$ are assigned the values from the DATA statement at line 50. At line 30, the values of the two variables are displayed.

A READ statement can accept data from a DATA statement that appears anywhere in a program. A DATA statement does not have to precede the READ statement in order to be effective.

REM

A REM statement is used to insert comments in a program. The REM statement is ignored by the BASIC interpreter.

Configuration

REM *remarks*

Example

```
REM INPUT ROUTINE
```

Any statements that follow a REM statement, on the same line, are also ignored by the computer. As a result, a REM statement is generally used on its own line or at the end of a multiple statement line.

RESTORE

A RESTORE statement is used to move the DATA statement pointer to the beginning of the DATA item list.

Configuration

RESTORE

The data in a program is read in order, starting with the first DATA statement item. In order to reread the data, a RESTORE statement is necessary.

When a RESTORE statement is executed, the next READ statement will assign to its first variable the first data value that appears in the program.

Example

```
]10 READ A1,B1,C1,X1$
]20 PRINT A1,B1,C1
]30 PRINT X1$
]40 RESTORE
]50 PRINT
]60 READ A2,B2,C2,X2$
]70 PRINT A2,B2,C2
]80 PRINT X2$
]90 READ X3$
]100 PRINT X3$
]110 DATA 32,-102,2.12,RECTOR,SPALL
```

In the preceding example, data is read into the variables indicated in line 10. The data is then displayed in lines 20 and 30. The RESTORE statement in line 40 allows the data items read in line 10 to be read again.

RESUME

RESUME is used in SmartBASIC to resume program execution after an ONERR GOTO statement has branched program control to an error handling routine.

Configuration

```
RESUME
```

If RESUME is executed without an error having previously occurred, the program will stop, the system will hang, or an error message will result.

RETURN

A RETURN statement is used to branch a program back to the line where the last subroutine was called.

Configuration

RETURN

A subroutine is called with a GOSUB or ON,GOSUB statement. When the subroutine has been completed, a RETURN statement causes program control to return to the statement following the most recently executed GOSUB or ON,GOSUB statement.

Example

RETURN

RIGHT$

The RIGHT$ statement is used to return the rightmost characters of a string.

Configuration

*b$* = RIGHT$ (*a$*,*c*)

The RIGHT$ function returns a string value. The first argument is a string constant or a string variable. The second argument is a numeric value. The string returned consists of the number of characters specified by the numeric argument. These characters are the rightmost characters in the string argument.

Example

```
]10 A$ = "WILLIAM JONES"
]20 PRINT A$
]30 PRINT RIGHT$(A$,5)
]RUN
WILLIAM JONES
JONES
```

The preceding example contains a program that uses a RIGHT$ statement. At line 10, the string variable A$ is assigned the value "WILLIAM JONES". At line 20, the value of A$ is printed. At line 30, the rightmost 5 characters of the value of A$ are displayed.

If the value of the numeric argument exceeds the length of the string argument, the entire string value is returned. If the value of the numeric argument is less than one or greater than 255, the following error message will be displayed:

```
?Illegal Quantity Error
```

RND

The RND function is used to generate "random" numbers.

Configuration

X = RND (*a*)

In SmartBASIC, RND will return a random number greater than or equal to zero and less than one. If RND's argument (*a*) is positive, a new random number will be generated each time RND is executed.

RND can also be used with a negative argument. The same random number will be returned when RND is executed with the same negative value for *a*.

If *a* = 0, RND will return the most recently generated random number.

If the same negative argument is repeated followed by RND statements with positive arguments, the same series of random numbers will be generated. This is illustrated in the following example.

Example

```
]100  PRINT RND(-1)
]200  PRINT RND(3)
]300  PRINT RND(.22)
]400  PRINT RND(.33)
]500  PRINT RND(-1)
]600  PRINT RND(.99)
]700  PRINT RND(2.7)
]800  PRINT RND(.77)
]RUN
.763671875
.0966934073
.805508261
.897540095
.763671875
.0966934073
.805508261
.897540095
```

ROT=

ROT= sets the amount of rotation for a shape which is to be plotted with DRAW or erased with XDRAW.

Configuration

ROT=*x*

x can range from 0 through 255. The shape will be rotated 90 degrees clockwise for every increment of 16 in the value of *x*. For example, ROT=0 causes the shape to be drawn in the same position in which it was originally defined. ROT=16 causes the shape to be rotated 90° clockwise. ROT=32 causes the shape to be drawn upside down. ROT=64 causes the shape to be drawn in its original position.

The number of actual different rotations is limited by the SCALE= setting. Lower SCALE= settings will have fewer noticeable rotations. When SCALE= is set to one, there are only eight noticeable ROT= values. They are 0,8,16,24,32,40,48,56 and numbers greater than 63 which would use the MOD64 equivalent. If a number other than 0,8,16,24,32,40,48, or 56 is used, the shape will generally be drawn with the lower corresponding ROT= value.

Example

```
ROT=32
```

The ROT statement given in our example would cause a shape plotted with DRAW to be rotated 180 degrees.

RUN

RUN is used to execute the BASIC program currently stored in memory. Prior to program execution, all variables, pointers, and stacks will be cleared.

Configuration

RUN [*linenumber*]

or

RUN *filename*

If *linenumber* is specified, execution will begin at the specified line. If no *linenumber* is specified, execution will begin with the lowest line number. If a non-existent *linenumber* is specified, one of the following error messages will result:

```
?Undefined Statement Error
```

When used in the second configuration, RUN looks for the file specified by *filename*, tries to load it, and if successful, executes it. If the specified file is not on the Digitial Data Pack, an error message is printed.

SAVE

The SAVE command is used to store a program on a Digital Data Pack.

Configuration

SAVE *filename* [,D*x*]

SAVE need only be entered with the program's *filename.* If the indicated *filename* duplicates that of a file already on the Digital Data Pack, the contents of the original program will be put into a backup file with the same name, and the original program will be replaced with the new program.

Example

```
]10 PRINT "HI THERE": REM EXAMPLE PROGRAM
]SAVE PROG
]CATALOG
Volume: FIRST DIR
 A      1 PROG
251 Blocks Free
]SAVE PROG
]CATALOG
Volume: FIRST DIR
   A     1   PROG ◄——————— new program
   a     1   PROG ◄——————— backup
  250   Blocks Free
```

SCALE

The SCALE command gives the size at which a shape is to be displayed. The SCALE command takes the following form.

Configuration

SCALE = *x*

x is a numeric argument with a range of 1-255. This command should be executed before DRAWing any shape to the screen. If *x* = 10 then the ADAM will draw the shape ten times larger than if *x* = 1.

Example

```
]10 HGR2
]20 HCOLOR = 9
]30 FOR I = 1 TO 32
]40 SCALE = I
]50 ROT = I
]60 DRAW 1 AT 128,96
]70 NEXT I
```

In the preceding example, line 10 initialized the screen so that graphics could be drawn. Lines 20 defined a color for the shape. Lines 30-70 set up a loop which rotated and enlarged the shape.

SCRN

SCRN is used in the low resolution graphics mode to return the color code of the point specified as its argument.

Configuration

SCRN (*x*,*y*)

x and *y* can range from 0 to 39. If *x* is in the range from 0 to 39, SCRN will return the color code of the point whose column is indicated by *x* and whose row is indicated by *y*.

Example

```
]10  GR
]20  COLOR = 2
]30  PLOT 20,10
]40  COLOR = 6
]50  PLOT 30,35
]60  PRINT SCRN(20,10), SCRN(30,35), SCRN(10,0)
]70  END
```

In the preceding example, line 10 initializes the low resolution screen. Lines 20 to 50 plot two different colored dots on the screen. Line 60 displays the color value of the corresponding dots. Notice that the value of SCRN(10,0) is zero. There is a black dot at that position.

SGN

The SGN function returns a +1 if its argument is positive, a -1 if negative, and a 0 if zero.

Configuration

```
SGN (a)
```

Example

```
]100  A = 100
]200  X = SGN(A)
]300  PRINT X
]400  END
]RUN
1
```

SIN

The SIN function returns the sine of the angle specified as its argument. The argument will be assumed in radians.

Configuration

X = SIN (*a*)

Example

```
]PRINT SIN (3.1415927/2)
1
```

SPC

The SPC statement is used to insert spaces in a PRINT statement.

Configuration

SPC (*a*)

The argument of the SPC statement specifiés the number of blank spaces that will occur.

Example

```
]10 X = 4
]20 Y = 6
]30 PRINT X;SPC(5);Y
]40 END
]RUN
4     6
```

In the previous example, the values of the variables X and Y are printed at line 30. The SPC statement within the PRINT statement causes the output to be separated by 5 extra spaces.

SPEED

The SPEED statement sets the speed at which characters are output.

Configuration

```
SPEED = x
```

x can range from 0 to 255, with 0 being the slowest speed and 255 the fastest.

SQR

SQR returns the positive square root of its argument.

Configuration

```
SQR (a)
```

Example

```
]10 X = 49
]20 PRINT SQR(X)
]RUN
7
```

STOP

The STOP statement causes a halt in the execution of a SmartBASIC program.

Configuration

```
STOP
```

If STOP is executed in the program mode, the following screen message will be displayed.

```
?Break in line
```

line is the line number of the executed STOP statement.

CONT can be used to resume program execution after it has been halted by a STOP statement.

Example

```
]10  INPUT X
]20  IF X = 10 THEN STOP
]30  PRINT X
]40  END
```

In the preceding example, if a value of 10 is input for X in line 10, the program execution will stop and the following message will be displayed.

```
?Break in 20
```

By entering CONT, program execution will resume with line 30.

STR$

STR$ returns the string representation of its argument.

Configuration

X$ = STR$(*a*)

Example

```
]10  A$ = STR$(40)
]20  PRINT A$
]30  END
]RUN
40
```

In the preceding example, the string variable A$ is assigned the string value "40". The STR$ function converts the numeric value 40, to the string value "40".

TAB

In SmartBASIC, the TAB function moves the cursor to the right to the column specified as its argument. TAB must be used with a PRINT statement.

Configuration

TAB (*column*)

TAB erases existing screen data as it moves to the right. If the specified *column* is not to the right of the current column position, the cursor will not move.

column can range from 1 to 255. If *column* is greater than the allowed for the output device will move down to the next output line where tabbing will continue.

Example

```
]10 X = 1:Y = 2
]20 PRINT
]30 PRINT X;
]40 PRINT TAB(48);
]50 PRINT Y
]60 END
```

In the preceding example, X is output at the leftmost column of the display line. TAB then moves the current print position to the middle of the next display line where Y is output.

TAN

TAN returns the tangent of its argument in radians.

Configuration

a = TAN(*b*)

Example

```
]10 A = TAN(35*3.141593/180)
]20 PRINT A
]RUN
.700207635
```

TEXT

TEXT returns the screen to the text mode from any of the graphics modes.

Configuration

```
TEXT
```

TEXT clears the screen, and homes the cursor.

TRACE

TRACE displays the line number of each statement as it is executed. Generally, TRACE is used as a debugging tool.

Configuration

```
TRACE
```

TRACE can be turned off by executing NOTRACE.

VAL

The VAL function converts its string argument to a numeric value. The numeric characters in the string argument will be converted to their numeric equivalents until an unacceptable string character is encountered. The acceptable characters consist of the digits (0-9), the decimal point, a leading plus or minus sign, blank spaces, and in scientific notation an additional plus or minus sign, the letter E (for exponent), and an additional decimal point.

If the first character encountered by VAL is an unacceptable character, a value of zero is returned.

Configuration

VAL (*a$*)

Example

```
]10 A$ = "1.731E+02"
]20 B$ = "+97.5"
]30 C$ = "57CA"
]35 D$ = "E59"
]40 PRINT VAL(A$)
]50 PRINT VAL(B$)
]60 PRINT VAL(C$)
]70 PRINT VAL(D$)
]RUN
 173.1
 97.5
 57
 0
```

VLIN

VLIN is used to draw a vertical line in low-resolution graphics.

Configuration

VLIN *row 1*, *row 2* AT *column*

A vertical line will be drawn at the specified *column* from *row 1* to *row 2*. The color of the line will be determined by that specified in the last COLOR statement executed.

row 1 and *row 2* must be in the range of 0 to 39. *column* must lie in the range 0 to 39. If a value outside of these ranges is used, the following error message will be displayed:

?Illegal Quantity Error

If VLIN is executed in the text mode or high resolution graphics mode, the statement will have no effect.

Example

```
]10 GR
]12 COLOR = 3
]15 FOR I = 1 TO 20
]20 VLIN 10,30 AT I
]25 NEXT I
]30 END
```

VTAB

VTAB moves the cursor to the indicated row on the screen. VTAB causes the cursor to move up and down but never sideways.

Configuration

VTAB *row*

row can range from 1 to 23. A *row* value outside of that range results in the following error message:

```
?Illegal Quantity Error
```

Example

```
]5 HOME
]10 VTAB 1 : PRINT
]20 VTAB 2:PRINT "ROW 2"
]30 VTAB 10:PRINT "ROW 10"
]40 VTAB 20:PRINT "ROW 20"
]50 END
```

XDRAW

XDRAW is used to erase a graphics shape in high resolution graphics.

Configuration

XDRAW *shape* [AT *column, row*]

The scale and rotation of the shape to be erased must be specified indentical to the scale and rotation of the shape when it was drawn. XDRAW effectively draws the shape as DRAW would, except that XDRAW always uses black as the default color, regardless of the present high resolution color.

Example

```
]10 HGR2
]20 SCALE = 9
]30 HCOLOR = 9
]40 FOR I = 0 TO 255
]50 ROT = I
]60 DRAW 1 AT 128,96
]70 XDRAW 1 AT 128,96
]80 NEXT I
```

Operating System Command

SmartBASIC can use the majority of the operating system commands. These commands are used in a program to facilitate data storage and retrieval. The format of these commands are as follows.

PRINT CHR$(4);"*command*"

By PRINTing the CHR$(4), SmartBASIC will be altered that the next characters to be printed represent an operating system command. Refer to Chapter 10 for specifics on the usage of the following operating system commands:

CATALOG
CLOSE
DELETE
INIT
LOAD
LOCK
MON
NOMON
OPEN
READ
RENAME
RUN
SAVE
UNLOCK
WRITE

OTHER COMMANDS

The SmartBASIC interpreter will accept a number of commands which seemingly have no useful programming purpose. These include:

FLASH
IN#
RECALL
STORE
WAIT

It is possible that these commands were included in SmartBASIC in order to make the language compatible with Applesoft BASIC.* Almost any Applesoft program can be entered into SmartBASIC and run without modification.

* FLASH, IN#, RECALL, STORE and WAIT are reserved words in Applesoft BASIC. Applesoft BASIC® is a registered trademark of the Apple Computer Co.

Appendix 1. SmartBASIC Reserved Words

ABS(
AND
ASC(
AT
ATN(
CALL
CHR$(
CLEAR
COLOR=
CONT
COS(
DATA
DEF
DEL
DIM
END
EXP(
FLASH
FN
FOR
FRE(
GET
GOSUB
GOTO
GR
HCOLOR=
HGR
HGR2
HIMEM:
HLIN
HOME
HPLOT
HTAB
IF
IN#
INPUT
INT(
INVERSE
LEFT$(
LEN(
LET
LIST
LOG
LOMEM:
MID$(
NEW
NEXT
NORMAL
NOT
NOTRACE
ON
ONERR
OR
PDL(
PEEK(
PLOT
POKE
POP
POS(
PRINT
PR#
READ
RECALL
REM
RESTORE
RESUME
RETURN
RIGHT$(
RND(
ROT=
RUN
SCALE=
SCRN(
SGN(
SHLOAD
SIN(
SPC(
SPEED=
SQR(
STEP
STOP
STORE
STR$(
TAB(
TAN(
TEXT
THEN
TO
TRACE
USR(
VAL(
VLIN
VTAB
WAIT
XDRAW

Appendix 2.
SmartWriter Quick Reference Guide

MOVE CURSOR

Description	Key Sequence
Right one character	→
Left one character	←
Up one line	↑
Down one line	↓
Up to top left of screen	HOME
Right to end of screen	HOME + →
Left to end of screen	HOME + ←
Up to top of screen	HOME + ↑
Down to bottom of screen	HOME + ↓

SAVE FILES

Description	Key Sequence
Store block	STORE-GET\STORE HI-LITE\DRIVE A\ filename\STORE HI-LITE
Store screen	STORE-GET\STORE SCREEN\DRIVE A\ filename\STORE SCREEN
Store workspace	STORE-GET\STORE WK-SPACE\DRIVE A\ filename\STORE WK-SPACE

FILE and BLOCK OPERATIONS

Description	Key Sequence
Get file directory	STORE-GET\GET\DRIVE A
Get file	STORE-GET\GET\DRIVE A\filename\ GET FILE
Get backup directory	STORE-GET\GET\DRIVE A\BACKUP FILE DIR
Make backup file	STORE-GET\STORE WK-SPACE\DRIVE A\ STORE WK-SPACE
Delete file	STORE-GET\GET\DRIVE A\filename\ DELETE\FINAL DELETE
Read file into text	STORE-GET\GET\DRIVE A\filename\ GET FILE
Print file	PRINT\PRINT WK-SPACE\PRINT
Hi-lite text	HI-LITE OFF\underline text
Erase hi-lite	HI-LITE ERASE
Move hi-lite	MOVE-COPY\MOVE\HI-LITE FIRST\ HI-LITE LAST\MOVE
Copy hi-lite	MOVE-COPY\COPY\HI-LITE FIRST\ HI-LITE LAST\COPY
Delete hi-lite	DELETE\HI-LITE\FINAL DELETE
Write hi-lite onto file	HI-LITE\STORE/GET\STORE HI-LITE\ DRIVE A\filename\STORE HI-LITE
Print hi-lite	PRINT\PRINT HI-LITE\PRINT

DELETE and INSERT

Description	Key Sequence
Delete character	BACK-SPACE
Delete hi-lite	DELETE\HI-LITE\FINAL DELETE
Delete file	STORE-GET\GET\DRIVE A\filename\ DELETE\FINAL DELETE
Insert text	INSERT\text\DONE
Insert file	STORE-GET\GET\DRIVE A\filename\ GET

FIND and REPLACE

Description	Key Sequence
Find text	SEARCH text START SEARCH/DONE
Find and replace text	SEARCH\text\START SEARCH\REPLACE\ text\REPLACE\DONE
Find and replace all	SEARCH\text\START SEARCH\REPLACE\ text\REPLACE ALL

SCREEN FORMAT

Description	Key Sequence
Standard to moving window	SCREEN OPTIONS\MOVING WINDOW
Moving window to standard	SCREEN OPTIONS\STANDARD FORMAT
Select white background	SCREEN OPTIONS\SELECT COLOR\WHITE
Select green background	SCREEN OPTIONS\SELECT COLOR\GREEN
Select black background	SCREEN OPTIONS\SELECT COLOR\BLACK
Select grey background	SCREEN OPTIONS\SELECT COLOR\GREY
Select blue background	SCREEN OPTIONS\SELECT COLOR\BLUE
Turn sound off	SCREEN OPTIONS\NO SOUND
Select partial sound	SCREEN OPTIONS\PARTIAL SOUND
Select full sound	SCREEN OPTIONS\FULL SOUND

PAGE FORMAT

Description	Key Sequence
Set left margin	MARGIN-TAB-ETC\HORIZ MARGIN\ LEFT 10\use cursor keys\DONE
Set right margin	MARGIN-TAB-ETC\HORIZ MARGIN\ RIGHT 76\use cursor keys\DONE
Set top margin	MARGIN-TAB-ETC\VERT MARGIN\TOP\ use cursor keys\DONE
Set bottom margin	MARGIN-TAB-ETC\VERT MARGIN\ BOTTOM\use cursor keys\DONE
Set tab	MARGIN-TAB-ETC\TABS\TAB SET
Clear tab	MARGIN-TAB-ETC\TABS\TAB CLEAR
Clear all tabs	MARGIN-TAB-ETC\TABS\ALL CLEAR
Increase line spacing	MARGIN-TAB-ETC\LINE SPACING\UP\ . . . \DONE
Decrease line spacing	MARGIN-TAB-ETC\LINE SPACING\ DOWN\. . . \DONE
Start new page	MARGIN-TAB-ETC\END PAGE
Letter to legal size	MARGIN-TAB-ETC\TYPE OF PAPER\ LEGAL 14
Legal to letter size	MARGIN-TAB-ETC\TYPE OF PAPER\ LETTER 11

PRINTING FORMAT

Description	Key Sequence
Single sheet to fan-fold	PRINT\PRINT OPTIONS\FAN FOLD
Fan-fold to single sheet	PRINT\PRINT OPTIONS\SINGLE SHEET
Superscript	SUPER-SUBSCRIPT\SUPERSCRIPT\text\ DONE
Subscript	SUPER-SUBSCRIPT\SUBSCRIPT\text\ DONE
Print hi-lite	PRINT\PRINT HI-LITE\PRINT
Print screen	PRINT\PRINT SCREEN\PRINT
Print workspace	PRINT\PRINT WK-SPACE\PRINT
Automatic page numbering	PRINT\PRINT OPTIONS\AUTO PAGE #
Set first page number to n	PRINT\PRINT OPTIONS\FIRST PAGE IS 1\. . .\FIRST PAGE IS n
Stop print	STOP PRINT
Re-start print after stop	PRINT

MISCELLANEOUS

Description	Key Sequence
Abort command sequence	ESCAPE-WP
Undo last command	UNDO
Print hi-lite	PRINT\PRINT HI-LITE\PRINT
Print screen	PRINT\PRINT SCREEN\PRINT
Clear screen	CLEAR\CLEAR SCREEN\FINAL CLEAR
Clear workspace	CLEAR\CLEAR WK-SPACE\FINAL CLEAR

Appendix 3.
ASCII Codes

This appendix outlines the characters associated with the ADAM's character set. By entering PRINT CHR$(X), where X is one of the following codes, the associated character will be displayed. Also, by entering PRINT ASC("Y") where Y is the character, the code associated with that character will be displayed.

Code	Character	Code	Character	Code	Character
1	°	31	⊿	61	=
2	☺	32	Space	62	>
3	⌂	33	!	63	?
4	♥	34	"	64	@
5	♦	35	#	65	A
6	♠	36	$	66	B
7		37	%	67	C
8	Backspace	38	&	68	D
9		39	'	69	E
10	Line Feed	40	(	70	F
11	Σ	41	)	71	G
12	Clear Screen	42	*	72	H
13	Carriage Return	43	+	73	I
14	✓	44	,	74	J
15	∞	45	-	75	K
16	Print Screen	46	.	76	L
17	≥	47	/	77	M
18	≤	48	0	78	N
19	±	49	1	79	O
20	♪	50	2	80	P
21	♩	51	3	81	Q
22		52	4	82	R
23	○	53	5	83	S
24		54	6	84	T
25	♩	55	7	85	U
26	♪	56	8	86	V
27	▶	57	9	87	W
28		58	:	88	X
29	┘	59	;	89	Y
30	┌	60	<	90	Z

Code	Character	Code	Character	Code	Character
91	[	104	h	116	t
92	\	105	i	117	u
93	]	106	j	118	‘v
94	∧	107	k	119	w
95	—	108	l	120	x
96	‘	109	m	121	y
97	a	110	n	122	z
98	b	111	o	123	{
99	c	112	p	124	|
100	d	113	q	125	}
101	e	114	r	126	~
102	f	115	s	127	Cursor
103	g				

Codes 164-255 correspond to the reverse of characters 36-127.

Index

$9.95 in USA • $12.95 in Canada

Coleco ADAM User's Handbook

The Coleco ADAM computer system has impressive computing and word processing capabilities. The **Coleco ADAM User's Handbook** provides clear, concise, and complete instructions which allow the user to master these capabilities.

The **Coleco ADAM User's Handbook** is written in a simple, concise manner so that even a first-time user can understand the Coleco ADAM, yet it still contains a wealth of advanced information for the experienced user. A complete guide to computing and word processing with the ADAM is included.

The following topics are covered in detail:

- Installation and Operation
- SmartWriter Word Processing
- SmartBASIC Programming
- Printer Usage and Maintenance
- Graphics
- Digital Data Drives
- Files and File Handling
- Reference Guide to SmartBASIC

The **Coleco ADAM User's Handbook** is a must for any user or potential user of the Coleco ADAM computer.

WSi WEBER SYSTEMS INCORPORATED

ISBN: 0-938862-45-6 LC: 84-2240