

User's Handbook to the **ATARI** 400/800® COMPUTERS



Jeffrey R. Weber
Stephen J. Szczecinski

USER'S HANDBOOK TO THE ATARI 400/800® COMPUTERS

by:

Jeffrey R. Weber
Stephen J. Szczecinski

Weber Systems, Inc.
Cleveland, Ohio

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

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

User's Handbook To The Atari 400/800® Computers
First Edition

Copyright© 1983 by Weber Systems, Inc.. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopy, recording, or otherwise without the prior written permission of the publisher.

Library of Congress Catalog Card Number 82-051088
ISBN 0-938862-15-4

Typesetting: Chelley Hoffman
Production & Design: Beth Cammarn

CONTENTS


1. INTRODUCTION TO THE ATARI COMPUTERS AND PERIPHERALS

7.

Atari 400 and 800 7. Atari 800 Specifications 9.
Atari 400 Specifications 10. Atari Keyboard 11.
Atari Video Display 11. Plug-In Cartridge
Hatches 13. Computer Memory 13. Atari 410
Program Recorder 14. Atari 810 Disk Drive 15.
Atari Printers 16. Atari 850 Interface Module 16.
Game Controls 17. Software 17. Operating
Systems 17. Languages 18. Applications
Programs 18.

2. INSTALLATION AND OPERATION OF ATARI COMPUTERS

19.

Introduction 19. Installing the Atari 400 21.
Installing the Atari 410 Recorder 21. Installing
the Atari 810 Disk Drive 21. Installing the Atari
820 Printer 23. Installing the Atari 822 Printer 24.
Installing the Atari 825 Printer 24. Installing a
ROM Cartridge 25. Power On 26. Keyboard 30.
System Reset Key 31. Select Key 31. Option Key
31. Start Key 31. Return Key 32. Break Key 32.
Shift Key 32. Ctrl Key 33. Caps/Lowr Key 33.  Key 34. Arrow Keys 34. Back S Key 35. Clear
Key 35. Insert & Delete Keys 35. Tab Key 35. ESC
Key 36. Auto Repeat 36. Display Line Length 36.

3. INTRODUCTION TO ATARI BASIC

37.

Immediate & Program Modes 37. Line Numbers 38. NEW 40. END 40. Program Execution 40. Program Lines & Display Lines 41. Multiple Statement Program Lines 41. Abbreviating Keywords 42. Listing a Program 42. Error Messages 43. BASIC Data Types 44. Floating Point Numbers 44. Scientific Notation 45. Tables & Arrays 49. Expressions & Operators 51. Compound Expressions & Order of Evaluation 52. Arithmetic Operations 53. Relational Operators 55. Logical Operators 56. Atari BASIC Statements 59. Remark Statements 59. Assignment Statements 60. DATA, READ 60. Outputting DATA 62. INPUT Statements 64. Loops 66. Nested Loops 67. Conditional Statements 68. Branching Statements 68. ON, GOTO 70. Subroutines & GOSUB Statements 70. ON, GOSUB 72. Break Key & CONT 72. System Reset Key 72. STOP 73. END 73. Atari BASIC Functions 74.

4. ADVANCED ATARI BASIC

75.

Atari ASCII 75. String Handling 76. Substrings 76. String Concatenation 77. CHR\$ & ASC Functions 78. Escape Sequences in Strings 79. Graphics Characters in Strings 80. Variable Storage 82. PEEK & POKE 83. Screen Output Programming 84. Using the Carriage Return in Cursor Positioning 84. TAB 85. Moving the Cursor With Escape Sequences 86. Home Cursor 87. POSITION 87. Changing the Display Screen Margins 88. Screen Input Programming 88. Prompt Messages 88. Input Response Checks 89.

5. ATARI BASIC REFERENCE GUIDE

91.

ABS 92. ADR 92. AND 92. ASC 94. ATN 94. BYE 95. CLOAD 95. CHR\$ 96. CLOG 96. CLOSE 97. CLR 97. COLOR 98. COM 102.

CONT 103. COS 104. CSAVE 104. DATA 105.
 DEG 106. DIM 107. DOS 110. DRAWTO 112.
 END 114. ENTER 114. EXP 115. FOR 116. FRE
 118. GET 119. GOSUB 123. GOTO 125.
 GRAPHICS 126. IF 126. INPUT 129. INT 132.
 LEN 133. LET 133. LIST 134. LOAD 136. LOCATE
 137. LOG 139. LPRINT 139. NEW 140. NEXT
 141. NOT 142. NOTE 143. ON 144. OPEN 145.
 OR 153. PADDLE 154. PEEK 155. PLOT 156.
 POINT 157. POKE 158. POP 159. POSITION
 160. PRINT 161. PTRIG 166. PUT 166. RAD 169.
 READ 170. REM 171. RESTORE 171. RETURN
 172. RND 172. RUN 173. SAVE 174. SETCOLOR
 175. SGN 175. SIN 175. SOUND 176. SQR 177.
 STATUS 177. STICK 178. STRIG 179. STOP 180.
 STR\$ 181. TRAP 181. USR 182. VAL 183. XIO
 184.

6. ATARI 410 PROGRAM RECORDER

189.

Introduction 189. Data Files 189. Program Files
 190. Saving Programs 190. Program Recording
 Formats 191. Loading a Program 192. RUN C:
 195. Reading and Writing Data 197. Opening
 Data Files 198. Closing Data Files 200. Writing to
 a Data File 200. Reading From Data Files 202.

7. ATARI 810 DISK DRIVE

205.

Types of Disks 205. Hard Disks 205. Winchester
 Disk Drives 206. Floppy Diskettes 207. Tracks &
 Sectors 208. Hard & Soft Sectors 209. Single &
 Double Sided Diskettes 211. Diskette Density
 211. Write Protection 212. Disk Files
 213. Filename Match Characters 213. Atari DOS
 215. Disk Buffer 217. Booting DOS 217. DOS
 Menu 218. Disk Directory 220. Run Cartridge
 222. Copy File 223. Delete File 227. Rename
 File 228. Lock File 230. Unlock File 231. Write
 DOS File 231. Format Diskette 232. Duplicate
 Disk 233. Binary Save 234. Binary Load 236. Run
 At Address 237. Create MEM.SAV 238.
 Duplicate File 239. Saving BASIC Programs 240.

Loading a Program 242. Chaining Programs 243. Opening a Disk File 244. Closing a Data File 246. Writing to a Data File 247. Reading From a Data File 248. NOTE and POINT 250.

8. ATARI PRINTERS

253.

LIST P: 253. LPRINT 254. PRINT# & PUT 255. Printer Buffer 255. Printer Character Sets 255. Atari 825 Control Characters 256. Line Feed 258. Reverse Line Feed 258. Half-Line Feed & Reverse Half-Line Feed 259. Carriage Return 259. Underlining 259. Standard, Condensed, & Proportionally Spaced Character Sets 260. Backspace & 1-6 Dot Spaces 260.

9. ATARI GRAPHICS & SOUND

263.

GRAPHICS 263. GRAPHICS 0 263. Color Registers & SETCOLOR 265. GRAPHICS 1 & 2 267. COLOR 272. PLOT 277. DRAWTO 278. GRAPHICS 3 thru 8 278. POSITION 281. LOCATE 282. PUT 283. XIO 283. Atari Sound 285.

APPENDIX A. Atari Error Messages

287.

APPENDIX B. Atari BASIC Reserved Words

294.

APPENDIX C. Atari ASCII Code Set

295.

APPENDIX D. Atari 400/800 Memory Map

301.

APPENDIX E. Atari PEEK & POKE Locations

306.

Index

315.

CHAPTER 1.

INTRODUCTION TO THE ATARI COMPUTER AND PERIPHERALS

Introduction

In this book, we will describe the Atari home computers as well as the peripherals that can be attached to them such as disk drives, cassette recorders, and printers.

In the first chapter of this book, we will discuss the features of the Atari 400 and 800 computers, the 410 Program Recorder, the 810 disk drive, game controls, and the various Atari printers. In the second chapter, we will discuss the installation and operation of the Atari 400 or 800 and its various peripherals.

In the third and fourth chapters, we will discuss programming the Atari in Atari's version of the BASIC programming language. The fifth chapter contains a reference guide to the various Atari BASIC commands, statements, and functions.

In Chapters 6, 7, and 8, we will discuss the Atari Cassette Recorder, Atari Disk Drive, and Atari printers in greater detail. In Chapter 9, we will discuss the usage of graphics and sound on the Atari 400 and 800.

Atari 400 and 800

There are two Atari computer models; the Atari 400 and the Atari 800 (pictured in Illustration 1-1).

The Atari 400 and 800 are very similar. Both models function the same and follow the same set of instructions. The difference between the Atari 400 and 800 lies in the fact that the 800 has features that the 400 does not.

For instance, the Atari 800's memory can be expanded, while the

memory of the Atari 400 is more or less fixed. Also, with the Atari 800, a video monitor can be used for video output as well as a regular television set. With the Atari 400, only a regular television set can be used for video output. A video monitor offers a more detailed picture than a regular television set. Also, the Atari 800 has a typewriter style keyboard while the Atari 400 has a flat panel with the keys outlined on it. Finally, the Atari 800 allows two accessory cartridges to be plugged in, while the Atari 400 allows only one.

However, the Atari 400 does have one major advantage--it costs less than the Atari 800.

From hereon, we will refer to both the Atari 400 and 800 collectively as the Atari, unless a distinction between the two is necessary. Whenever we refer to one model, the reader can assume that the concept applies to the other model as well, unless we specify otherwise.

Illustration 1-1. Atari 800 Computer



Atari 800 Specifications

The Atari 800 consists of a group of components which include the following:

- Computer Console
- TV Switch Box
- AC Power Adapter
- Atari 800 Operator's Manual
- Atari BASIC Manual
- Atari BASIC Language
- Atari Educational System

The Atari 800 Console contains the central processing unit or CPU, the operating system in ROM, 8K or 16K of RAM, and two expansion slots for additional RAM. The Atari 800 console also contains the keyboard, 2 cartridge slots, controller jacks, and a serial I/O port.

The TV switch box allows a regular TV set to be used as the Atari's video display. The AC power adapter converts regular AC current to a low voltage that can be used by your Atari. The AC power adapter can be plugged into any normal household outlet.

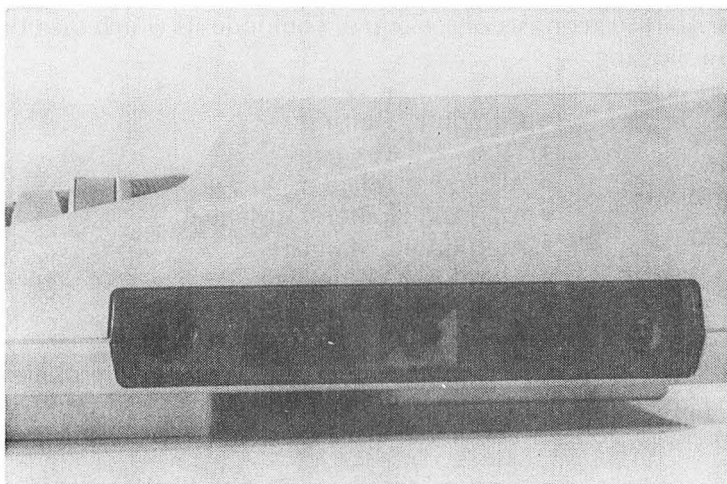
The Atari Educational System and Atari BASIC language are contained in two cartridges. Operator's instructions are included with each of these.

The Atari 410 program recorder allows the use of programs which have been stored on cassette tape. The Atari 410 also allows the user to save his programs from RAM onto cassette tape for later use.

The Atari 800's software is known as the operating system and is contained on a 10K ROM cartridge. The operating system controls the entire flow of information within the computer.

As shown in Illustration 1-2, the side panel of the Atari 800 contains several switches and jacks. The monitor jack can be used to connect a video display monitor or a video tape

Illustration 1-2. Atari 800 Side Panel



recorder to your Atari 800.

The Atari 410 Program Recorder, Atari 810 disk drive, and Atari 820 printer are all installed by plugging into the peripheral jack. More than one device can be connected through the peripheral jack via a daisy chain configuration, where all devices to be used are connected together. This is explained in more detail in Chapter 2.

The 2-3 channel switch should be set to the same channel as the television set being used for video output. Use channel 2 or 3, whichever has the poorer reception.

The Atari 800 contains four controller jacks located in the front of the console beneath the keyboard. These can be used for connecting game controllers or a light pen.

Atari 400 Specifications

Your Atari 400 includes the Atari 400 console, as well as a TV switch box, operator's manual, and an AC Power Adapter.

The Atari 400 console contains the CPU, operating system in ROM, 8K or 16K of RAM, one cartridge slot, controller jacks, and one I/O connector.

The TV switch box allows a regular TV set to be used for video output for the Atari 400. The AC power adapter converts household current to a low voltage that can be used by the Atari.

Atari Keyboard

The Atari keyboard allows the user to interact with the computer. The instructions entered at the keyboard are transferred to the computer. The keys on both the Atari 400 and 800 are arranged in the same order as on a regular typewriter. However, the Atari keyboard contains several special keys not found on a standard typewriter keyboard. These keys will be discussed in Chapter 2.

As mentioned in the previous section, the Atari 800's keyboard features a typewriter style keyboard with raised keys. The Atari 400 keys are identified on a flat panel on the front of the unit.

Atari Video Display

Generally, a home color television set is used as the video display screen for the Atari. A black and white television set can also be used, in which case, the different colors will appear as various shades of gray.

The Atari 800 allows the use of a video monitor as well as a television set as its video display unit. A video monitor (either color or black & white) tends to cause images to be displayed in greater detail than a television set.

A television set is connected to the Atari computer with a switch box that is itself connected to the television's antenna terminal. This is shown in Illustration 1-3. The switch box has two positions. One position allows the set to be used with the Atari, while the other allows the set to function as a television.

If a video display monitor is being connected to the Atari 800, a

switch box is not necessary. This connection can be accomplished by attaching the 5-pin plug into the socket on the side of the Atari 800. This is shown in Illustration 1-4.

Regardless of whether a television set or a monitor is being connected to the Atari, several different modes of display are available. One of these is the **monochromatic text mode**. This mode is used to display one color plus white (ex. black and white, blue and white, etc.). In the monochromatic text mode, the screen is divided into 24 lines of 40 characters each. Two other modes are available for displaying text in up to four different colors. Other modes are available for displaying graphics. These will be discussed in detail in Chapter 9.

Illustration 1-3. Atari/Television Set Hook-Up

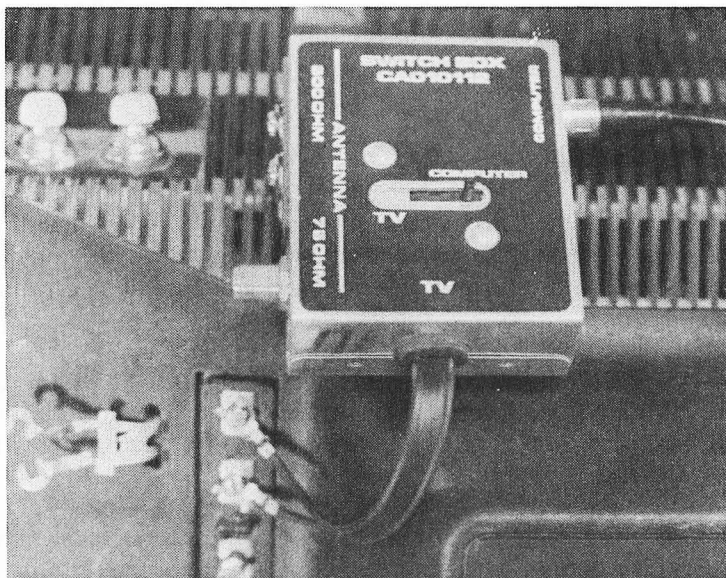
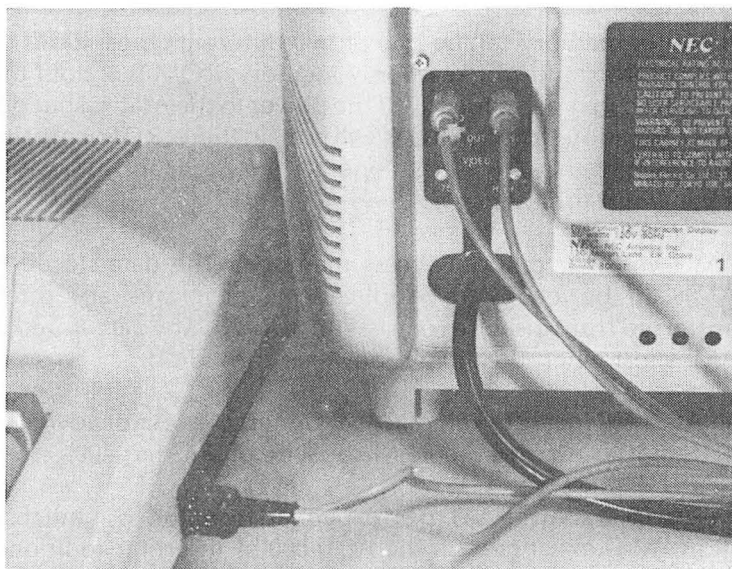


Illustration 1-4. Atari 800/Video Display Monitor Hook-Up



Plug-In Cartridge Hatches

Both the Atari 400 and 800 have a hatch on the top of the unit which can be opened for the purpose of inserting a plug-in cartridge (see Illustration 1-1). These cartridges contain ROM memory (discussed later) on which programs are stored. These programs may be games, applications programs, or even a BASIC language interpreter.

The Atari 400 allows the insertion of a single cartridge while the 800 allows two cartridges to be inserted.

Computer Memory

Computer memory is measured in units known as **bytes**. A byte is used to store a single character in the computer's memory. Bytes are represented in units of measurement known as

kilobytes or **K**. 1K is the equivalent of 1024 bytes. Your Atari may contain from 18 to 60K of memory (or 18,432 to 61,440 bytes).

Computer memory can be one of two different types; **ROM** or **RAM**. ROM stands for read-only memory. ROM will hold the data stored in it permanently. If the power to the Atari is shut off, the information stored in ROM will remain there. ROM contains the programs that are used to operate the Atari, and allow it to interact with the user.

RAM stands for random-access memory*. The data stored in RAM can be changed. Applications programs are often transferred from diskettes or cassette to RAM. Any data stored in RAM is lost when the Atari's power is turned off.

The Atari 400 includes 16K of RAM. Generally, it is not advisable to attempt to expand the RAM capacity of an Atari 400.

The Atari 800 allows RAM to be expanded from 16K to as much as 48K. RAM is expanded on the Atari 800 by inserting additional RAM plug-in modules underneath the unit's top cover. Expanding the Atari's RAM is explained in more detail in Chapter 2.

Atari 410 Program Recorder

Cassette tape can be used to store programs in RAM and then transfer these programs back into RAM at some later date. The Atari 410 Program Recorder (as shown in Illustration 1-5) is designed for use with the Atari computer. Approximately 50K or 51,200 bytes of data can be stored on a 30 minute cassette.

*Random access memory is a somewhat misleading term to describe, RAM, as most memory (including ROM), is randomly accessed.

Illustration 1-5. Atari 410 Program Recorder



Atari 810 Disk Drive

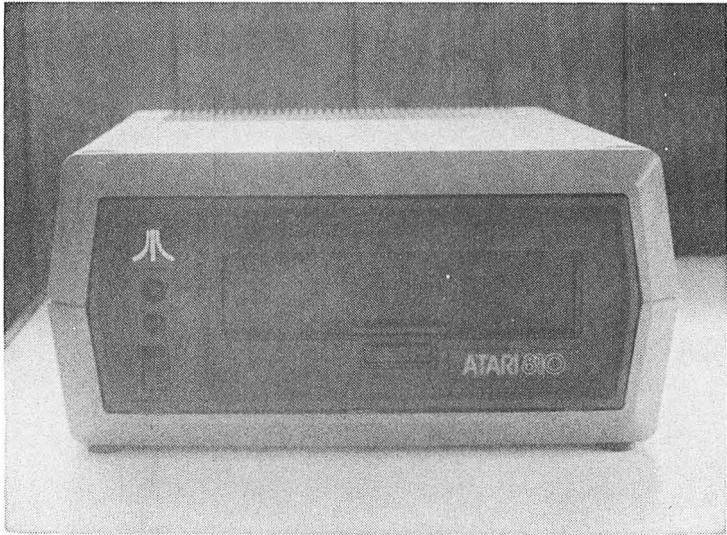
A disk drive is a much more efficient device for storing data than a cassette recorder. A disk drive allows greater storage capacity, quicker access to data, as well as fewer errors in data transfers.

The Atari 810 disk drive (as shown in Illustration 1-6) is designed to be used with Atari computers. The Atari 810 uses single-sided single density diskettes.

Diskettes which are designed to be written on only one side are known as single sided (SS) diskettes. Diskettes designed to be written on both sides are known as double sided (DS) diskettes.

Density refers to a diskette's recording format, which in turn affects its capacity. Single-sided single density diskettes (as used with the Atari 810) have a capacity of approximately 94K.

Illustration 1-6. Atari 810 Disk Drive



The Atari 810 disk drive can only be used with the Atari 800 computer with a minimum of 16K of RAM.

Atari Printers

Atari produces three different printers; the 820 Printer, 822 Thermal Printer, and 825 Wide Carriage Printer.

The 820 and 822 Printers are connected to the Atari computer via the I/O Data Channel. The 825 Printer is connected to the Atari computer with the 850 Interface Module. The 850 Interface Module can be used to connect printers other than the 825 to the Atari.

Atari 850 Interface Module

The Atari 850 Interface Module allows communications between

the Atari computer and RS-232-C peripherals. We already discussed the fact that the Atari 825 printer should be connected via the Atari 850 Interface Module.

Another Atari peripheral that must be connected via the Atari 850 Interface Module is the Atari 830 Modem. The Atari 830 Modem allows your Atari to communicate with another terminal also equipped with a modem over telephone lines.

The Atari 850 Interface Module is connected to the Atari console. In turn, the peripherals are connected to the 850 Interface Module. The 850 Interface Module has 4 serial ports and 1 parallel port, known as the printer port. The 850 Interface Module has its own memory and processor and is programmed from the Atari computer.

Game Controls

Three types of game control devices can be used with the Atari; joysticks, paddles, and keyboard controllers.

Software

Software can be described as the instructions or programs that cause the computer to operate. Several different classifications of software exist for the performance of different functions. These can be classified as operating systems, languages, and applications programs.

Operating Systems

An operating system can be defined as a group of programs which manage the overall operation of the computer. The operating system performs system operations such as the loading and unloading of data from cassette or diskette into RAM and the display of operator keyboard entries on the video screen.

The Atari's operating system is stored permanently in ROM. The operating system is contained in a plug-in module in the Atari 800.

Languages

Programs are generally written in a high-level language that is different from the instructions the computer uses. A program known as an **interpreter** must be used to translate the high-level language into a form that the computer can comprehend.

BASIC is the high-level language generally used with the Atari. The Atari BASIC interpreter is contained on a ROM cartridge which can be plugged in under the hatch of either the Atari 400 or 800.

Applications Programs

Applications programs are those written to accomplish a specific task. Examples of applications programs are games, word processing programs, financial forecasting programs, and accounting programs. Generally, applications programs are stored on cassette or diskette and are transferred into RAM, where the program is available to the computer.

Applications programs for the Atari can also be stored in a permanent form on a ROM cartridge. This ROM cartridge can be plugged in underneath the hatch on the Atari. Examples of ROM plug-in cartridges are shown in Illustration 2-4.

CHAPTER 2.

INSTALLATION AND OPERATION OF ATARI COMPUTERS

Introduction

If you are a first-time computer user, your Atari may seem a little confusing at first. However, using a computer is really very simple. In this chapter, we hope to show you exactly how simple your Atari is by showing you step-by-step how to install and operate it.

Installing the Atari 800

First of all, when you unpack your Atari 800, save the carton and packing material. These should be used if the Atari is to be moved or stored.

The Atari 800 is easy to install. First of all, install either a video monitor via the monitor jack on the side of the console or a TV set using the TV switch box.

The TV switch box has been designed so that it can be permanently installed on your TV set, as it allows regular TV reception as well as video output for the Atari. The TV switch box has an adhesive backing that can be used to attach it to the back of your TV set.

The TV switch box contains a switch marked Computer/TV. When this switch is at the Computer position, the TV set receives its signals from the Atari 800. When the switch is set to the TV position, the TV set receives its signals from your television antenna.

To install the TV switch box, first of all, disconnect your television antenna from the VHF terminals at the back of your TV set. The antenna should be either of the following:

- 75 OHM with screw-on connector
- 300 OHM with two flat leads

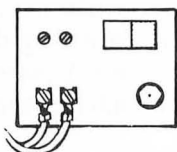
Attach either the 75 OHM or 300 OHM connector to the matching connector on the side of the TV switch box.

Next, attach the 300 OHM connector (with the two flat leads) leading from the bottom of the TV switch box and labeled TV, to the VHF terminals on your TV set.

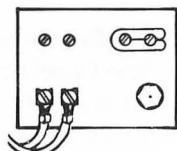
If your television antenna is a 300 OHM model, the TV switch box installation is finished. If your antenna is a 75 OHM model, you must convert your television to accept a 300 OHM signal from the TV switch box.

Refer to Illustration 2-1. If the antenna box contains a switch as shown in the top drawing, just push the switch to the 300 OHM position. If the antenna box resembles that shown in the middle drawing, loosen the screws holding the U-shaped slider, and move it to the 300 OHM position. If the antenna box resembles the last drawing, screw the round wire into the connector as pictured.

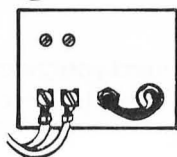
Illustration 2-1. Television Set Conversion to 300 OHM



If your TV set resembles this drawing, push the switch to the 300 SL position.



If your TV set resembles this drawing, loosen the screws and move the slider to the 300 SL position.



If your TV set resembles this drawing, screw the rounded wire into the connector.

Once the TV switch box has been connected, plug the AC power adapter into any ordinary 115V household outlet. Plug the end of the AC power adapter into the power jack on the side of the Atari console. Then, follow the power-on procedures as described later in this chapter.

Installing The Atari 400

The installation procedures for the Atari 400 are virtually identical to those for the Atari 800. Follow the steps just outlined for Atari 800 installation if you are installing an Atari 400.

Installing The Atari 410 Program Recorder

The Atari 410 Program Recorder is packaged with a power card and a peripheral data card, which is permanently attached to the recorder.

Use caution when using the Atari 410. Do not use the Atari 410 outdoors. Also, do not allow liquids to be spilled on the Atari 410, or allow it to be dropped in water.

The first step in installing the Atari 410 is to plug the data card (which is permanently attached to the 410) to the peripheral jack on the side of the Atari's console. Next, plug the recorder's power card into the AC jack on the side of the recorder, and plug it into a household outlet.

Installing The Atari 810 Disk Drive

The Atari 810 will include the following:

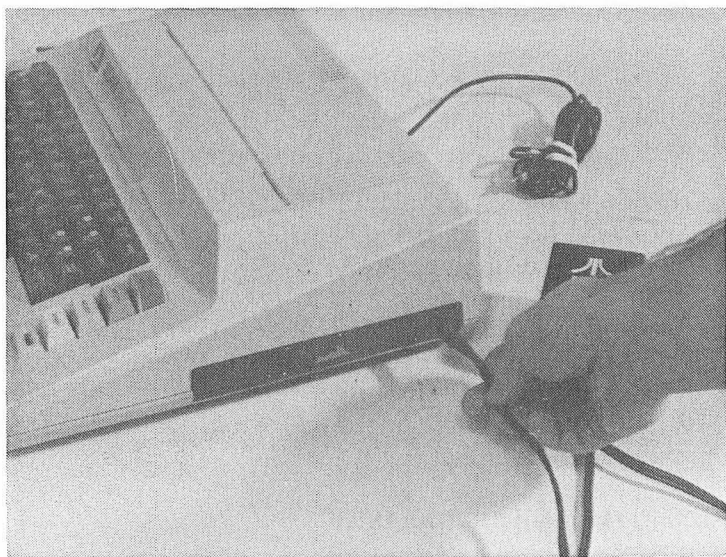
- 810 Disk Drive
- Data Cord (round card with identical end plugs)
- AC Power Adapter
- Owner's Manual
- DOS Diskette

Save the 810's carton and packing material, should the unit need to be moved or stored.

Before installing the Atari 810 disk drive, be certain that the power switches on both the Atari 810 and the computer are off.

The first step in installing the Atari 810 is to plug one end of the AC power adapter into a household outlet, and the other to the Atari 810 console. This is shown in Illustration 2-2.

Illustration 2-2. Installing the Atari 810



Next, plug one end of the data cord to the peripheral plug on the Atari console, and the other to one of the I/O connectors on the rear of the Atari 810. This is shown in Illustration 2-2. Additional peripherals can be connected via the unused I/O connector.

If just one disk drive is being installed, the device code switch in the back of the Atari 810 should be set to 1. If 2, 3, or 4 drives are to be installed, the switches should be set as indicated on the drive code diagram on the back of the Atari 810. This is shown in Illustration 2-5. Use a pen or screwdriver to move the switches to the appropriate setting. Be certain that the power to the Atari 800 and 810 is off when setting the drive code switch.

Installing the Atari 820 Printer

The Atari 820 Printer includes the following items:

- Printer
- Roll of Paper
- Paper Mandrell
- Ribbon
- Data Cord
- User's Manual
- Attached Power Card

Never operate a printer without the ribbon and paper installed. Doing so may cause damage to the printing head solenoids. The instructions for loading the ribbon and the paper in the Atari 820 are given in the operator's manual.

Once the Atari 820 has been loaded with paper and a ribbon has been installed, plug the power cord attached to the unit into a household outlet.

Next, plug one end of the data cord into the port labeled 'peripheral' on the Atari computer console. If another peripheral such as the Atari 810 Disk Drive has already been installed via the peripheral port, the Atari 820 can be connected via the I/O CONNECTOR port on the Atari 810 disk drive. Plug the other end of the data cord into either of the I/O CONNECTOR ports on the printer.

The printer is now installed. Turn the printer's power switch on and press the paper advance button once. The printer is now ready for paper to be loaded.

Installing the Atari 822 Printer

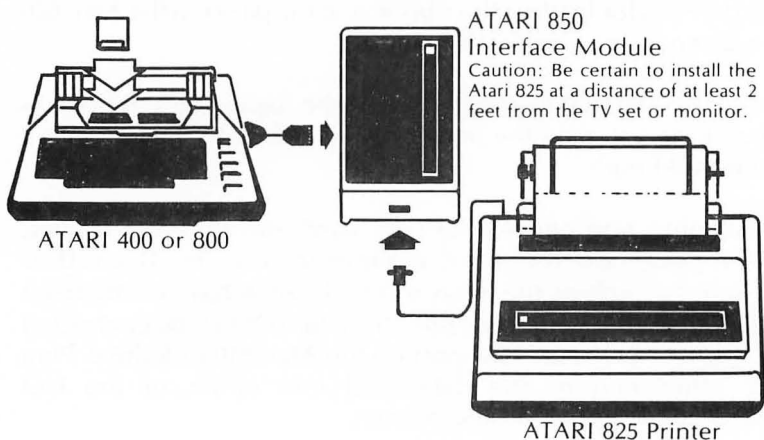
The installation procedure for the Atari 822 printer is essentially the same as for the Atari 820 printer.

Installing the Atari 825 Printer

The Atari 850 Interface Module is required to install the Atari 825 printer to either the Atari 400 or 800. The Interface Module converts serial data from the computer into parallel data used by the Atari 825.

The installation procedure for the Atari 825 is depicted in Illustration 2-3. A few words of caution are in order before beginning installation. First of all, the Atari 825 should be installed at a distance of at least 2 feet from your television set. Secondly, be certain that all of the power switches on both the Atari 825 and Atari 400 or 800 are turned off prior to installation. Finally, note that the Atari 825 is delivered with a ribbon installed. During installation, try to keep the Atari 825 level. Otherwise, the ribbon may fall out of its tray.

Illustration 2-3. Installing the Atari 825 Printer



Once these precautions have been taken, use an Atari I/O Data Cord to connect the Atari 400 or 800 to the 850 Interface Module. Connect an AC adapter to the Power In jack on the 850 Interface Module. Connect the other end to a regular household AC outlet.

Connect the 3 prong power cord on the Atari 825 printer to an outlet. The Edge-on connector of the Atari printer cable should be connected to the printed circuit card connector on the back of the printer. The side of the connector marked 'This side up' should be facing up when the connection is made. Do not attempt to force this connector, as this could damage the cable connector. Connect the other end of the printer cable to the parallel bit printer interface connection on the 850 Interface Modules.

The 850 Interface Module must be turned on before the Atari 825 can be used. Programming procedures for the Atari 825 will be covered in Chapter 8.

Installing a ROM Cartridge

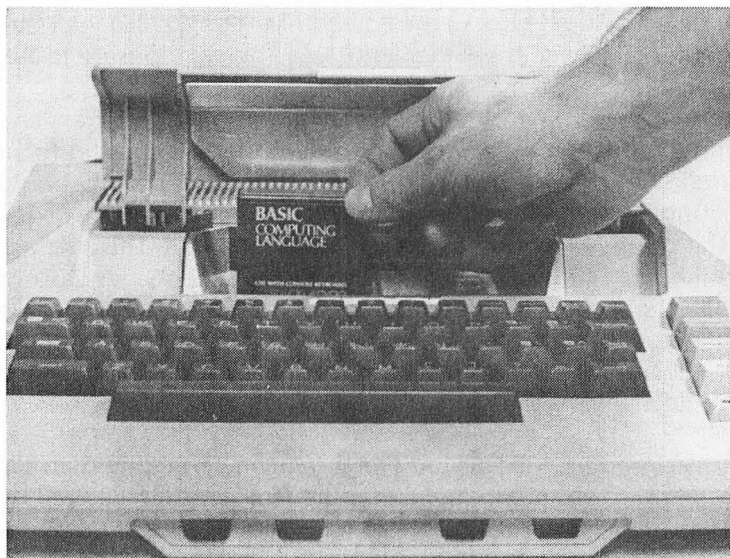
As discussed in Chapter 1, the ROM cartridges are installed under the hatch cover on the top of the Atari. The Atari 400 has one socket, while the Atari 800 has two.

Generally, cartridges are installed in the left slot. When inserting a cartridge, hold it so that its label is facing towards you. Plug the cartridge into the socket and press it all the way into the socket. Finally, close the hatch. This is shown in Illustration 2-4.

When the Atari is operated without a cartridge installed, it will be operating in the **memo pad mode**. In this mode, all the Atari can do is display what has been entered at the keyboard. Obviously, the memo pad mode is not very useful.

In our discussions in this book, we will assume that the BASIC Computing Language ROM cartridge is installed.

Illustration 2-4. Installing a ROM Cartridge



Turning on the Power

Once your Atari system has been properly installed, you may turn on its power. Use the following procedure in turning on the various components of your Atari system.

1. Turn on the television or monitor. If you are using a television set, be certain both the set and the Atari are both turned to the same channel. The switch connected to the television set should be placed on computer.
2. If you are using the Atari 810 disk drive, turn on drive 1 and insert a diskette with the Atari disk operating system (DOS) on it. Close the drive door once the diskette has been inserted.
3. If a serial device that has been connected to the 850 Interface Module is to be used, turn on the 850. Otherwise, leave it turned off.

4. Turn on the Atari 400 or 800 console unit.
5. Turn on the printer when you wish to use it. Remember, if you are using the 825 printer, the 850 Interface module must also be turned on.

Unless the preceding power-on procedure is followed, the Atari may not be able to interact with some of the system components.

Step 1. Turning on the Television

First of all, turn on the television set or monitor, whichever your system is using. If you are using a monitor, you can skip the remainder of Step 1 and proceed to Step 2.

If you are using a television set, first of all be certain that the switch that is connected to your television's antenna terminal is set to computer. Tune in your set to channel 2 or 3, whichever is weaker in your area.

The Atari computer must be set to broadcast on the same channel that the television is tuned to. This is accomplished with the switch on the side of the Atari (see Illustration 1-2). Set this switch so that it corresponds to the television channel used (2 or 3).

Step 2. Turning on the Disk Drive

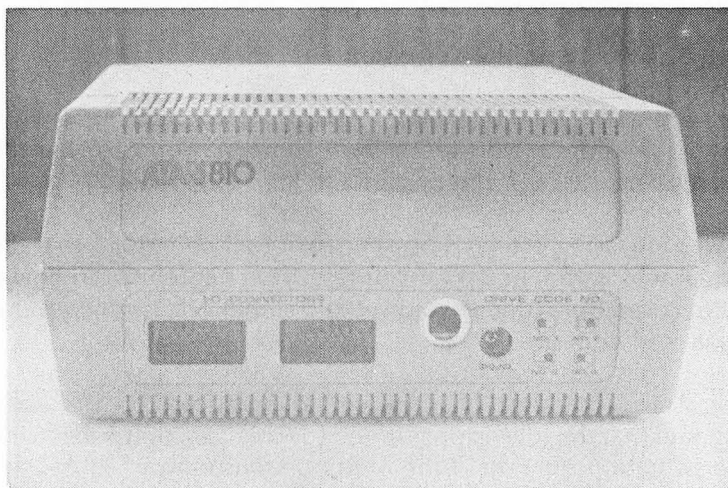
If your system does not include a disk drive or if the disk drive will not be used, you need not turn it on and can skip to Step 3.

If the disk drive is to be used, turn on drive 1. When turned on the drive will emit whirring and clicking sounds for a few seconds, and the lights on its front panel will light. The sounds will soon stop, and all lamps except for the power lamp will go off.

If your Atari system contains more than 1 drive, only drive 1 needs to be turned on at this point. By examining the access hole in the back of the 810 drive (see Illustration 2-5), the user can determine which is drive 1.

The access hole will contain one or two switch levers. The position of these levers determines the drive number. Drive 1's levers are both positioned to the left. Only the black lever in front may be visible, at it may be hiding the white lever which is situated behind it.

Illustration 2-5. Determining the Disk Drive Number



Once you have determined which drive is drive 1, insert either the 'Disk File Manager Master Copy', a 'Disk File Manager II Master Copy', or a copy of one of these in that drive. The label side of the diskette should be facing up. Slide the diskette all the way in and close the door behind it.

Step 3. Turning on the 850 Interface Module

The 850 Interface Module only needs to be turned on if a device is attached to it. If not, you can leave it off.

Step 4. Turning on the Atari 400/800

You now are ready to turn on the heart of your Atari system--the Atari 400 or 800 computer. First of all, be certain that the correct ROM cartridge has been installed, and that all system

components have been properly connected.

Now, locate the power-on switch on the side of the console as shown in Illustration 1-2. Turn the switch to the on position, and turn up the volume on your television set a little.

The power lamp on the keyboard should come on. Also, your television set will begin making noises, and a blue field with a black border will be displayed. If the disk drive is on, it will begin to whirl.

Finally, the message, **READY**, will be displayed in white letters on the screen, and the disk drive will stop whirling.

If the **READY** message is not displayed within 3 seconds, a problem exists somewhere in the system. Be certain the components of your system are properly connected, and that the proper ROM cartridge is in place. Repeat the start-up procedure. If the Atari still does not start, call your dealer for assistance.

If the following message appears on the display:

BOOT ERROR

the problem probably lies with the disk drive. Be certain that a DOS diskette has been installed label side up, and that the disk drive door is closed.

Step 5. Turn on the Printer

Once Steps 1 through 4 have been accomplished, the printer may be turned on as desired. Of course, printing operations can not be undertaken unless the printer is on. Remember, the 825 printer requires that the 850 Interface module be on.

The Ready Message

Once the Ready message appears on the display, the Atari computer is ready to accept commands entered by the user via the keyboard. Just beneath the **READY** message, a white square

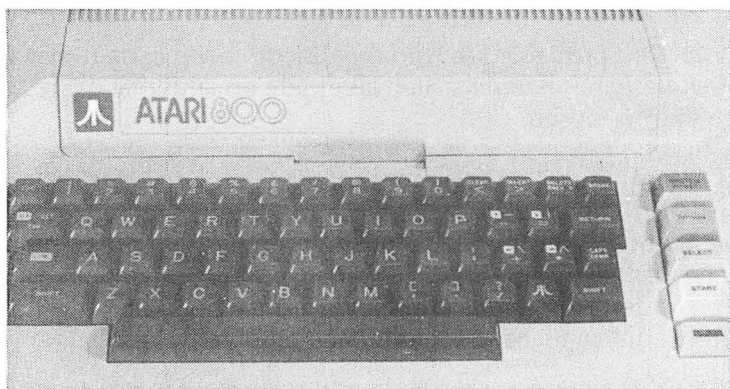
known as the cursor will be displayed. The cursor indicates the position where the next character typed in will appear on the display.

Atari Keyboard

As mentioned in Chapter 1, the Atari 400 and 800 keyboards are virtually identical, except that the Atari 800 keyboard contains a typewriter style keyboard with raised keys while the Atari 400 keyboard is depicted on a flat panel. The keyboard layout of the Atari keyboard is shown in Illustration 2-6.

The Atari keyboard contains many of the same keys arranged in the same order as a regular typewriter keyboard. The Atari keyboard also contains several additional keys not found on a typewriter keyboard. Two of these, ESC and CTRL, are located on the left side of the keyboard. Three other keys, BREAK, CAPS/LOWR, and M , are located on the right side of the keyboard. Also, to the far right of the keyboard are four yellow special function keys. Finally, some of the standard typewriter keys contain special words or special symbols.

Illustration 2-6. Atari Keyboard



In the next 17 sections, we will discuss the usage of all of the keys on the Atari keyboard. We recommend that you experiment with these keys as you read these sections. Do not worry about damaging the computer. Any error situation caused by keyboard entries can be corrected by merely turning the Atari off and then on again.

System Reset Key

Each of the four keys located to the right of the keyboard allows the user to select a different starting position within a cartridge.

The System Reset key is located at the top of the yellow function keypad at the far right of the keyboard. When the System Reset key is pressed, all computer operations stop, and control is restarted from the beginning of a cartridge.

Be careful not to press System Reset accidentally. Doing so can cause the loss of data--especially if the disk drive is in use when System Reset is pressed.

Select Key

Pressing the select key allows the user to view the initial screen at the start of the next game or program. In other words, the initial screen is 'selected'.

Option Key

The option key is pressed to record the user's choice of one of a number of options within an application program or game.

Start Key

The Select and Option keys are generally used to display a screen and record the user's choice. The next step is for the user to press the start key. This begins the action selected.

Return Key

As characters are entered via the keyboard, these characters are displayed on the video screen and also saved in memory. However, these characters are not actually interpreted by the computer until the Return key has been pressed. The Return key tells the Atari that the line into which characters are being typed has been finished.

When Return is pressed, the Atari will review the line just entered for errors. If any errors are found, an error message will be displayed.

Break Key

The Break key will stop any action being undertaken by the computer. For example, if you press Break while entering a BASIC command line, the computer will ignore all data entered on the current line.

Pressing Break may or may not affect a program depending upon how that program is written. Some programs are written so that pressing Break has no effect, while other programs may stop if Break is pressed. Generally, if a program is interrupted by pressing Break, it can be continued by typing in the BASIC command `CONT` and then pressing Return. However, the display screen will most likely be erased if Break is pressed during program execution.

Shift Key

Upon start-up, the keys for the letters (A-Z) always produce upper case letters on the Atari, regardless of whether the Shift key is depressed or released. However, the position of the Shift key does have an effect on many of the other keys on the Atari keyboard.

The keys affected by the position of the Shift key include two characters. The bottom character is output when the Shift is off (Unshift), and the top character is output when the Shift is on (Shift).

In this book, we will denote a key produced in the Shift mode by using the word Shift followed by the symbol or name of the character produced in Unshift. For instance, Shift 9 would denote the symbol (. The characters produced in the Shift mode are listed in Appendix C.

Ctrl Key

Ctrl is an abbreviation for the word 'control'. We will use Ctrl and Control interchangeably in this text.

The Control key is used in combination with another key much as the Shift key is. The Control key must be held down at the same time as the other key.

The use of the Control key with another key will be symbolized by prefixing the name of that key with Ctrl-. For example, Ctrl-C designates pressing the Control and C keys simultaneously.

Like the shift key, the Control key gives the key it is used with a different interpretation. Control is used with the letter keys to output the graphics characters. Control is used with many of the other keys to instruct the computer to undertake a particular function. For example, Ctrl+ results in the cursor being moved one space to the right. The various control key functions are listed in Appendix C.

Caps/Lowr Key

As mentioned earlier, upon start-up, the keys for the letters (A-Z) always produce upper case or capital letters, regardless of whether the Shift key is depressed or released. The Caps/Lowr key allows both capital and lower case letters to be output.

To output both capitals and lower case letters, press the Caps/Lowr key. When the Shift key is released, lower case letters will be output. When the Shift key is depressed, upper case letters will be output.

By pressing the Shift key and the Caps key simultaneously, the Atari will again output upper case letters.

The keyboard can be placed in the graphics character mode by pressing the Control and Caps/Lowr keys together. The graphics characters are pictured in Appendix C.

↵ Key

The ↵ key is used to switch the keyboard between the normal and the reverse video modes. In the reverse video mode, characters are displayed in blue on a white background.

Arrow Keys

The arrows keys will be referred to as follows in this text.

Up Arrow	→	Ctrl -
Down Arrow	→	Ctrl =
Left Arrow	→	Ctrl +
Right Arrow	→	Ctrl *

As you can see, the arrow keys are actually Control key combinations.

The arrow keys are generally used to move the cursor on the screen, so that keyboard entries can be corrected where necessary.

The Right and Left Arrow keys move the cursor to the right or left by one position along the same display line. These do not erase the characters that they pass over from the display. When the Right Arrow key is pressed with the cursor at the far right edge of a display line, the cursor will move to the left edge of the same line. When the left Arrow key is pressed with the cursor at the far left side of the display, the cursor will move to the far right side.

The Up and Down Arrow keys move the cursor up or down by one line. If the cursor is at the top of the screen, Up Arrow places the cursor at the bottom of the screen. If the cursor is at the bottom line of the screen, Down Arrow places it at the screen's top.

Back S Key

The Back S key moves the cursor one position to the left each time it is pressed. The character beneath the cursor is erased when Back S is pressed.

When the cursor is at the left edge of the screen and Back S is pressed, the cursor will not move.

Clear Key

Either the Shift< or Ctrl< key combination can be used to clear the display screen and move the cursor to the **home** position. The home position is the upper left-hand corner of the screen.

Insert and Delete Keys

Characters can be inserted or deleted by using the Control or Shift keys in combination with the >/Insert and Back S/Delete keys. Ctrl> results in a blank space being inserted to the right of the cursor. Ctrl Back S results in the character to the immediate right of the cursor being deleted. The cursor does not move when either Ctrl> or Ctrl Back S are pressed.

Shift> results in a blank line being inserted above the line that the cursor currently is in. The remainder of the display below the line the cursor is in moves down by one line.

Shift Back S causes the line that the cursor is currently in to be erased from the screen. The lines beneath that line are shifted upward in the display by one position.

Tab Key

When the Tab key is pressed, the cursor will move forward to the next tab position on the screen. Standard tab positions occur after every eight positions. The left margin on the Atari is indented two columns from the screen's edge. Because of this, the first tab stop occurs at the sixth position from the left margin.

Additional tab positions can be set by pressing Shift Tab at the

position desired. Pressing Ctrl Tab clears the tab stop at the cursor's current position.

ESC Key

ESC is an abbreviation for Escape, a term originally used with teletypes. The ESC key allows a key sequence to be entered in a program, without that sequence being executed as a function. ESC is always pressed and released prior to the entry of the key sequence whose effect is to be negated. This entry of ESC followed by the key sequence is known as an **escape sequence**.

For example, the following escape sequence,

ESC Ctrl<

would cause the display not to be cleared when Ctrl< is entered.

Other Atari Keys

The remaining Atari keys are used like those on a standard typewriter.

Auto Repeat

Atari's auto repeat feature functions with every key except Shift, Break, and System Reset. Auto repeat means that when a key is continuously pressed, that character will be repeated. For example, if the A key is pressed, a single A will be displayed on the screen. After a few seconds, the A will be repeated on the display as long as the A key is depressed.

Display Line Length

The Atari's display width is 40 characters. As mentioned earlier, the leftmost 2 characters comprise the left margin. Therefore, only 38 character positions are usable per display line.

CHAPTER 3. INTRODUCTION TO ATARI BASIC

Introduction

BASIC is probably the most widely used language in microcomputers, with the Atari being no exception. Atari BASIC is available in the ROM cartridge labeled "BASIC Computing Language".

To use Atari BASIC, you must have the Atari BASIC ROM cartridge. Also, you must have followed the correct start-up procedure as outlined in Chapter 2. The READY message will be displayed on the video screen when the Atari is ready to accept BASIC commands.

Immediate & Program Modes

The immediate mode is also known as the direct or the calculator mode. In the immediate mode, any BASIC command entry results in the instructions being executed without delay. For example, if the following immediate mode line was entered,

```
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 is executed when the appropriate command (generally RUN) is entered.

Illustration 3-1 contains an example of the entry of a program in the program mode and its execution.

Illustration 3-1. Program Mode Entry & Execution

READY

NEW

READY

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

READY



Line Numbers

In the program mode, program lines must begin with a line number. A line number is a one through five digit number entered at the beginning of a program line. The line number at the beginning of a program line is the only difference between it and an immediate mode line.

No two line numbers can be the same. If the same line number is used more than once in a program, the line most recently entered will replace the original. Line numbers can range from 0 to 32767.

The execution sequence of a BASIC program is determined by the value of its line number. The lowest line numbers will be executed first, followed by program lines with higher line numbers. Even if program lines are not arranged in sequential order, the Atari interpreter will place the lines in the correct order.

Adding program lines to a program stored in the Atari's RAM is very easy. Just type in the line number followed by the program line. The line will be inserted in the program in the position indicated by its line number. For example, by adding the following line,

```
35 PRINT "216-777-5579"
```

to the program in Illustration 3-1, the phone number for Jim Smith will be displayed on the line following his city, state, and zip.

Program lines can be deleted by typing the line number to be deleted followed by Return. For example, the following entry,

```
30 
```

would result in line 30 being deleted.

Program lines can be changed by merely retyping the new line. The existing line in the Atari's memory will be replaced with the new line. For example, the following entry,

```
10 PRINT "Thomas Hill"
```

would result in "Thomas Hill" being output rather than "Jim Smith" in the program in Illustration 3-1.

Program lines also can be changed by displaying them on the

screen with the LIST statement. Once that line has been listed to the screen, it can be edited using the cursor control keys as described in Chapter 2.

Once the desired changes have been made, these must be made permanent. This is accomplished by pressing the Return key while the cursor is within that line. Unless the Return key is pressed somewhere within the line being edited, any changes made effect only the video display. The cursor can be positioned anywhere within the program line when Return is pressed.

NEW Command

You may have noticed the execution of the NEW command in Illustration 3-1. The NEW command is used to erase an old program from memory before a new one is typed in.

The Atari can only store one program in RAM at any one time. If you attempt to enter a new program while another program is already stored in RAM, the new program will be merged with the existing program.

END Statement

Notice the last line in the program in Illustration 3-1. That line consists only of the line number plus the BASIC reserved word END.

The END statement identifies the end of a program, and instructs Atari BASIC to return to the immediate mode. Obviously, the END statement should be the last line in your program.

Actually, Atari BASIC does not require an END statement. When the program's final statement is executed, it will end. However, it is good programming practice to end a BASIC program with the END statement.

Executing a Program

A program is executed in the program mode by entering the RUN command. This is shown in Illustration 3-1. Every time RUN

is executed, the program is re-executed. As previously discussed, in the immediate mode, each program line is executed when the Return key is pressed.

Program Lines & Display Lines

A **display line** can be defined as one row on the video display. A **program line** is regarded by the BASIC interpreter as one line, regardless of the number of display lines it occupies on the screen. The end of a program line is signaled when the Return key is pressed.

Program lines generally are limited to 114 characters. If you are entering a lengthy program line, the Atari will beep when the 107th character has been input. This is intended as a warning to the operator that he is approaching the limit of the program line.

Multiple Statement Program Lines

A **statement** can be defined as an instruction to the computer. The terms statement and command are often used interchangeably. Most programs consist of a large number of statements. The following are examples of statements.

```
PRINT "Tim Gregory"  
070 DIM A(15)  
100 C = 2*B
```

Every statement in Atari BASIC must contain at least one **key** or **reserved** word. A keyword identifies the calculation, decision, input, or output function to be performed. The keywords are described individually in Chapter 5 and are listed in alphabetical order in Appendix B.

In addition to keywords, numeric constants, string constants, variables, and special symbols may appear in a BASIC statement. These are known as the statement **parameters**.

Atari BASIC allows the user to place more than one statement on a single program line. Multiple statements must be separated with a colon (:). The following is an example of a multiple

statement program line.

```
100 A = B * 7:PRINT A:PRINT B
```

Abbreviating Keywords

Many of the Atari BASIC keywords can be abbreviated. For example, the keyword PRINT can be abbreviated with the symbol "?". Generally however, keywords are abbreviated with a single letter or several letters followed by a period. For example, the keyword GOTO can be abbreviated as follows.

G.

The various abbreviations for the keywords are contained in Appendix B.

Listing a Program

As mentioned earlier, the LIST command can be used to display program lines currently stored in RAM. Remember, if the NEW command is issued or if the Atari is turned off, the program in RAM will have been erased, and can no longer be displayed by LIST.

LIST is used with the following configuration,

```
LIST (line 1, line 2)*
```

where *line 1* is the line number of the first line to be listed, and *line 2* is the line number of the last line to be listed.

LIST can be used without any parameters to list the entire program. LIST can also be used with a single line number to list just that program line.

*In this chapter, a standard format will be used to describe BASIC keyword configurations. The keyword will be displayed in our regular type style in upper case. Parameters will be displayed in our italic type style in lower case. Optional parameters will be enclosed in parentheses.

Error Messages

When the Atari encounters a statement with an error, an **error message** will be displayed. The error message consists of the following.

ERROR- *message*

message can be the statement causing the error or a diagnostic error message number. These error numbers are listed in Appendix A.

BASIC Data Types

Data can be classified under two major categories: **text** and **numeric**. Text data consists of characters. These characters are generally used within strings.

Examples of numeric data include:

Integers
Floating Point Numbers
Scientific Notation

Each of these data types will be discussed in the following sections.

Strings

A **string** consists of one or more characters enclosed within double quotation marks. The following are examples of strings:

"F. Scott Fitzgerald"
"149 Lexington Ave"
"New York, NY 10017"
"212-349-9879"

Notice that a string can contain both letters, numbers, and symbols. Any string containing numbers can not be used in a mathematical operation, unless it is first converted into numeric data. String to numeric data conversion is covered in Chapter 4.

Numeric Data

Atari BASIC stores all numbers in memory in **floating decimal point** form. With floating decimal point numbers, a decimal point is always assumed. Any number of digits can be placed on either side of this decimal point. Even with numbers with no decimal position, a decimal point always is assumed following the number's last digit.

Commas may not be included within numeric data. For example, 109000 would be a valid number in Atari BASIC, while 109,000 would be invalid.

Integer

An **integer** is a number without a decimal position. Integers can either be positive or negative. The following are examples of integers:

-1134
0
1
-1
17945
+32

Negative integers are preceded with the (-) sign. Positive integers can be preceded with the (+) sign, although integers without a (+) sign are assumed to be positive.

In Atari BASIC, integers are processed exactly as are any other floating point numbers. Atari BASIC does not process integers as a separate form of numeric data.

Floating Point Numbers

Floating point numbers include both integers, as well as decimal functions and numbers with decimal positions. The following are examples of floating point numbers.

```

-.0789
  5
77.39
  0
+.000001
67.98

```

Again, negative floating point numbers should be preceded with the minus sign (-). Positive floating numbers can optionally be preceded with the plus sign (+), however, a floating point number is assumed positive if it doesn't have a sign.

Scientific Notation

Atari BASIC uses **scientific notation** to express either extremely large or extremely small numbers. A number in scientific notation takes the following format:

$$\pm x E yy$$

Where;

\pm is an optional plus or minus sign.

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

E stands for exponent

yy is a one or 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 examples specify a number in both standard floating point and scientific notation:

```

1000000 → 1 E6
.000001 → 1 E-6
57500000 → 5.75 E+07
-.00000479 → 4.79 E-06

```

Any numbers containing more than 10 digits will be expressed in scientific notation. Also, any decimal number which contains more than two digits to the right of the decimal point will be expressed in scientific notation.

Atari BASIC can only handle floating point numbers expressed in scientific notation in the range between $-9.99999999 \text{ E}+97$ and $9.99999999 \text{ E}+97$. Any decimal numbers that are closer to zero than $+9.99999999 \text{ E}-98$ or $-9.99999999 \text{ E}-98$ will be converted to 0.

Rounding

In Atari BASIC, floating point numbers can have at most 9 significant digits. Any digits beyond 9 are replaced with zeros, beginning with the least significant digit.

The following examples give the values used by Atari BASIC for floating point numbers containing more than 9 digits.

17898743214798 \rightarrow 1.78987432 E+13
 -879836341832 \rightarrow 8.79836341 E+11
 7005.32144587931 \rightarrow 7005.32144

Fractional numbers in the range between 1 and -1 also may contain a maximum of nine digits. However, with numbers in this range, the nine significant digits are counted beginning with the first non-zero digit to the right of the decimal point.

The following examples give the values used by Atari BASIC for floating point numbers in the range between 1 and -1 which contain more than 9 digits.

.87547983621 \rightarrow 0.874579836
 .12789478987432187 \rightarrow 0.127894789
 -.478947821765789 \rightarrow 0.478947821
 .000000001407936579463 \rightarrow 1.40793657 E-09

BASIC Variables

So far, we have only discussed data **constants**. A constant can be defined as a fixed value. The following are examples of string and numeric constants.

```
"Jack Novet"  
"375"  
27.59  
0  
100000
```

A name can be used to express data as well as a constant. **Variables** are used to express data as a name.

A variable can be defined as a quantity that can assume any one of a group of values. Variables are represented by variable names. These consist of a letter followed optionally by additional letters and/or numbers. The value assumed by a variable is subject to change, depending upon the program statement being executed. For example, in the following,

```
100 LET A = 5.0  
200 LET B = 7.0  
300 LET A = A + B
```

the variable A is initially assigned a value of 5.0 and B is assigned a value of 7.0. In line 300, the variable A is assigned a new value equal to the sum of variables A and B, which is 12.0. The previous value of A is erased.

Note the use of the LET statement in the preceding example. The LET statement is used to assign a value to a variable. Whenever a LET statement is used in a program, the value of the variable on the left side of the equation is to be 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
```

BASIC Variable Names

Atari Basic allows any group of up to 114 characters to be used as a variable name--as long as the first character of the group is a capital letter of the alphabet, and as long as the variable name does not duplicate a reserved word (see Appendix B). Examples of reserved words are:

LET, GOTO, IF, READ, DATA

The following are examples of valid BASIC variable names,

A	JOHN
B23456	N4N
TOTAL.DATA	B%
A2	N

while the following are invalid variable names:

2BB7	END
1A	FOR
PRINT	COS

All of the preceding examples of valid variable names should be used to represent numeric data. Variable names can also be used to represent string data. These are known as **string variables**. String variable names consist of a valid variable name followed by the dollar sign (\$). The following are examples of valid string variable names.

```
A$
B1P$
A7$
```

Before a string variable can be used in a program, it must first be dimensioned with the DIM statement. If a string variable is not dimensioned before it is used in a program, the error 9 will occur.

A string variable is dimensioned by giving its name and its maximum size after the reserved word DIM. The maximum size must be enclosed in parentheses. The following DIM statement,

```
100 DIM A$(5)
```

dimensions a five character string. More than one string variable can be dimensioned in a single DIM statement. For example, the following DIM statement,

```
100 DIM A$(10), B$(5), C$(7)
```

would dimension 3 string variables.

Tables & Arrays

Earlier in this chapter, we introduced the concept of variables. A variable is designed to hold a single data item--either string or numeric. However, some programs require that hundreds or even thousands of variable names be used.

Obviously, the use of thousands of individual variable names could prove extremely cumbersome. To overcome this problem, BASIC allows the use of **subscripted 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),..., A(100)
```

Note that each subscripted variable is a unique variable. In other words, A(0) differs from A(1), A(2), A(3), A(4), etc.

Subscripted variables should be visualized as an array (or table). In our previous example, the data contained in the array defined by A would consist of one row with 101 columns in it. Such an array is a single-dimension array.

An array can also consist of two dimensions. Such an array is known as a two-dimensional array (or table). An example of an array of 4 rows and 3 columns is shown in Illustration 3-2.

A two-dimensional array contains two subscripts. The first subscript contains the row location, while the second subscript contains the column location. The subscripted variable $A(1,0)$ identifies the darkened area in the array shown in Illustration 3-2.

Illustration 3-2. Two-Dimensional Array

		Columns		
		0	1	2
Rows	0			
	1			
	2			
	3			

In the Atari BASIC, arrays can be used to represent numeric data. String arrays cannot be used in Atari BASIC.

Before any array variable can be used in a program, the size of that array must have been defined so that BASIC can reserve a memory area for it. This is also accomplished with the DIM statement. A single dimension numeric array with 11 variables could be defined with the following DIM statement:

DIM A(10)

Remember that array subscripts begin with 0. Therefore, the numeric array A which was dimensioned in the preceding statement, would have space reserved for the 11 array elements, not 10.

More than one array can be defined with a single DIM statement. This is shown in the example below:

```
100 DIM Z(5,2), B(100), C(2,3)
```

A DIM statement must appear in a program before the array variable it is dimensioning appears. If an array variable is used in a program before it is dimensioned, error 9 will occur.

Expressions and 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 ^ 2
14 ≤ 21
X AND Y
```

Atari BASIC includes several types of expressions including **arithmetic**, **relational**, and **Boolean**. In our previous examples, the first three examples are arithmetic expressions, while the fourth and fifth are 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 phrase describing the operation to be undertaken is known as the **operator**. The operators in our previous example were as follows:

```

+
+
^
≤
AND
```

The constants or variables which are affected by the operator are known as **operands**.

Compound Expressions and Order of Evaluation

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) AND (C + D)  
IF A = 1 AND B = 1 THEN C = 1
```

With compound expressions, it is necessary that the computer knows which operation should be undertaken first. Atari BASIC follows a standard order of evaluation within compound expressions. This order is outlined in Table 3-1.

Note that parentheses have the highest precedence level. In other words, any expression enclosed within parentheses will be evaluated first. If more than one set of parentheses appears in an expression, these will be evaluated from left to right.

One pair of parentheses can be used to enclose an operator enclosed within another set. In such an instance, Atari BASIC will evaluate the expression within the innermost set of parentheses first, followed by the next innermost set, etc.

When expressions have the same order of evaluation, they will be evaluated in order from left to right within the compound expression.

Table 3-1. Order of Evaluation

	Operator	Description
Parentheses	()	Used to alter order of evaluation.
Arithmetic Operators	^ - * / + -	Exponentiation Unary Minus Multiplication Division Addition Subtraction
Relational Operators	= < > < > <= >=	Equal To Not Equal To Less Than Greater Than Less Than or Equal To Greater Than or Equal To
Boolean Operators	NOT AND OR	Logical Complement Logical AND Logical OR

Arithmetic Operations

The symbols used for addition, subtraction, multiplication, division, and exponentiation are known as **arithmetic operators** in BASIC. The symbols + and - are used for addition and subtraction respectively. The asterik (*) is used to indicate multiplication, while the slash (/) is used to indicate division.

When a + or - sign precedes a number, the symbol is used to specify that number's sign. When + or - is used to change a

number's sign, that usage is known as a **unary** operation. Unary operators can be used to change the sign of a numeric constant or variable as shown below:

$$100 \text{ LET } A = -A$$

When unary operators are used in the manner shown above, the unary operation is regarded as an arithmetic operation.

The term **arithmetic expression** is used to describe the use of an arithmetic operator with numeric constants and/or variables. The following are examples of arithmetic expressions.

$$\begin{aligned} X + Y + 70 \\ 100/A + B \\ 3000 * 10 + 1 \end{aligned}$$

Exponentiation is the process of raising a number to a specified power. For example, in the following,

$$A^5$$

the numeric variable A would be evaluated as:

$$A * A * A * A * A$$

In BASIC, exponentiation is indicated with the caret symbol, \wedge .

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

$$8 * 3 + 7 \wedge 2$$

The preceding expression would evaluate to 73.

Relational Operators

The following relational operators are used in Atari BASIC.

<	→	less than
<=	→	less than or equal to
>	→	greater than
>=	→	greater 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 Atari BASIC, a value of 1 represents a condition of true, while a value of 0 represents false.

The only values returned by a comparison in BASIC 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 → 0 (false)
3 = 3 → 1 (true)
2<>2 → 0 (false)
(2=2) * 4 → 4
(1>7) + 7 → 7

```

The first three examples are easy enough to understand. In the fourth example, the relational expression (2=2) is evaluated first as true or 1. This result is then multiplied by 4 with a product of 4 as the result. In the fifth example, the relational expression (1 > 7) evaluates as false or 0. This result is added to 7, with the result being 7.

Relational operations using numeric operations are fairly straightforward. However, relational operations using string values may prove confusing to the first-time computer user.

Strings are compared by taking the ASCII value for each character in the string one at a time and comparing the codes.

If the strings are of the same length, then the string containing the first character with a lower code number is the lesser. If the length of the strings are unequal, then the shorter string is the lesser. Blank spaces are counted and have an ASCII value of 32.

The following comparisons between strings would all evaluate as true.

```
"ABC"="ABC"
"ABC ">"ABC"
"aAA" >"AAA"
"Alfred" <"Zachary"
A$ < Z$ where A$ = "Alfred" and Z$ =
"Zachary"
```

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

Logical Operators

Logical or Boolean operations are generally used in Atari BASIC 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 are best explained with a 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, she would be acting under the NOT logical operation. The NOT, AND, and OR logical operators are summarized in Illustration 3-3.

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 ELSE GOTO 300
B = -1:PRINT NOT B
```

In the first example, the result of the logical operation will be true if variable X has a value greater than 10 or if variable Y 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 value 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. Otherwise, program control will branch to line 300.

In the third example, B is set to a value of -1 (true). The value of NOT B is then printed. This will be 0 or false.

Illustration 3-3 contains tables that may prove of help when evaluating program statements using logical operators in combination with relational operators.

Illustration 3-3. Logical Operators

NOT Operation

T	F	A Operand
F	T	NOT A

AND Operation

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

OR Operator

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

Atari BASIC Statements

In the next several sections, we will discuss many of the more commonly used statements in Atari BASIC. These include the following:

- Remark Statements
- Assignment Statements
- Output Statements
- Input Statements
- Loops
- Conditional Statements
- Branching Statements
- Subroutines
- STOP, END Statements
- Atari BASIC Functions

Remark Statements

Remark statements are used to include a programmer's comments within a program. It is good programming practice to include numerous Remark statements in your programs. Not only do Remark statements make your programs easier for others to understand, they also help you remember your program's logic.

Remark statements consist of a line number, the reserved word REM, and the programmer's comment. An example of a Remark statement is given below.

```
100 REM Initialize I to 0
```

Remark statements are ignored by the Atari BASIC interpreter, but are included in program listings.

In multiple line statements, the REM statement must be the final statement. The Atari BASIC interpreter ignores all text following the keyword REM.

REM can be abbreviated as R. or with the period (.).

Assignment Statements

Assignment statements were discussed briefly earlier in this chapter. Assignment statements are used to assign values to variables. The following are examples of assignment statements.

```
100 LET A = 7
200 B = 42
300 NAME$ = "Phil"
400 X=1:Y=2:Z=3
```

Notice that the keyword LET is optional. Generally, LET is assumed. Both string and numeric variables can be assigned values with an assignment statement. Also, multiple assignment statements can be included in a single line, as long as each of the individual statements is separated by a colon.

DATA, READ Assignment Statements

Assigning values to a large number of variables with individual assignment statements could prove very cumbersome. The DATA, READ statements can be used to assign values to a large number of variables. The following is an example of a DATA, READ statement.

```
100 DATA 100, 500, 1000, "Jack"
200 READ A, B, C, D$
```

The DATA statement creates a list of constant values known as a DATA list. The items in the DATA list are assigned sequentially to the variables in the READ statement. A DATA list is depicted in Illustration 3-4.

Illustration 3-4. DATA List

100 DATA 100, 200, 300, 400, 500

...

400 DATA Monday, Tuesday, Wednesday,
Thursday, Friday

500 READ A, B, C, D, E

600 RESTORE

700 READ F, G, H, I, J

...

900 READ A\$, B\$, C\$, D\$, E\$

DATA List

400,600 →	A	F	100
	B	G	200
	C	H	300
	D	I	400
	E	J	500
500,700 →	A\$		Monday
	B\$		Tuesday
	C\$		Wednesday
	D\$		Thursday
	E\$		Friday
900 →			

400,600 → DATA list pointer position after the execution of lines 400 and 600.

500,700 → DATA list pointer position after the execution of lines 500 and 700.

900 → DATA list pointer position after execution of line 900.

DATA statements may contain numeric or string values. These values must be separated or **delimited** with commas. DATA statements may appear at any point in the program. No other statements can appear in the same program line with a DATA statement.

The DATA list uses a pointer to indicate which value within the list is to be assigned to the next variable in a READ statement. Before the first READ statement is encountered, the DATA list

pointer will point at the first value in the DATA list. As values from the DATA list are assigned to variables in the READ statement, the pointer will move sequentially to each successive item in the DATA list.

The values from the DATA list must match the type of variable to which they are assigned in the READ statement. In other words, a string value can not be assigned to a numeric or vice versa.

The RESTORE statement is used to reset the DATA list. In Illustration 3-4, note the use of the RESTORE statement. After DATA list values have been read into A, B, C, D, and E in line 500, a RESTORE statement is executed. This causes the DATA list pointer to be reset to the beginning of the DATA list.

Outputting Data

In some of our preceding examples, we touched upon the use of the PRINT statement to display data. The PRINT statement can be used to display both numeric and string data.

The following program statement,

```
100 PRINT "Vendor List"
```

would display the following at the current cursor position.

```
Vendor List
```

The first item in a PRINT statement is displayed at the cursor's current location.

Several strings can be displayed on the same line with a single PRINT statement by separating the string constants or variables in the PRINT statement with commas. The following statements,

```
050 DIM A$(10)
100 LET A$ = "John"
200 PRINT A$, "Bill", "Peter"
```

would result in the display shown below:

John	Bill	Peter
------	------	-------

Atari BASIC divides the spacing on a line into a series of print zones. Each print zone contains 10 spaces. 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. In our example above, John would begin in column 1 (print zone 1); Bill in column 11 (print zone 2); and Peter in column 21 (print zone 3).

A semicolon can also be used to separate the items in a PRINT statement. A semicolon causes the next item in the PRINT statement to be displayed immediately after the preceding item. Unlike the use of the commas in a PRINT statement, when semicolons are used to separate items, no blank spaces appear between the items when they are displayed.

When a PRINT statement has finished execution, the cursor moves to the left margin of the following line. This is known as a **carriage return/line feed**.

If a comma or semicolon occurs at the end of a PRINT statement, the carriage return/line feed will be suppressed. If a comma is placed at the end of the PRINT statement, the next PRINT statement will begin output at the next print zone after the last item is displayed. If a semicolon is placed at the end of the PRINT statement, the next PRINT statement will begin output immediately following the last item displayed.

In this section, we have only discussed sending output to the video display. Output can also be sent to the printer. This is accomplished by using the LPRINT statement in place of PRINT. The LPRINT statement is used exactly as the PRINT statement in Atari BASIC.

However, the LPRINT statement does have some variations when it is used with the Atari 825 Printer. These variations occur when a comma or semicolon is used to end the PRINT statement. If an LPRINT statement is used to print more than 40 characters,

any subsequent LPRINT statements will be started on a new line on the Atari 825.

However, if an LPRINT statement prints 38 characters or less and ends with a comma, output from any subsequent LPRINT statement will be begun on the same line at print position 41. Printing also begins at this position, if LPRINT is used to print 40 characters or less and ends with a semicolon.

INPUT Statements

Data can be input into the computer while a program is being executed. This is accomplished with the INPUT statement. For example, when the following statement is executed,


```
100 INPUT A
```

the computer will display a question mark and wait for the operator to enter a response. That entry will be assigned to the variable A. The entry must be ended by pressing the Enter key. Program execution will then resume.

The values of several numeric variables can be input with a single INPUT statement as shown in the example below.

```
200 INPUT X, Y, Z
```

When the preceding INPUT statement is executed, the INPUT prompt (?) will be displayed. The operator should then enter the data items for X, Y, and Z. 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.

```
100, 200, 300 
```

The INPUT statement in Atari BASIC functions somewhat differently with string inputs than with numeric inputs.

First of all, the string variable used with INPUT must have been dimensioned earlier in the program.

Secondly, the number of characters entered in response to the INPUT prompt cannot exceed the number of characters that the string variable specified in INPUT was dimensioned for. For example, in the following,

```
100 DIM A$(5)
200 INPUT A$
```

the string variable A\$ is only dimensioned for 5 characters in line 100. If the operator attempts to enter a string greater than 5 characters in response to the INPUT prompt in line 200, Atari BASIC will ignore any additional characters.

Finally, if string variables are included as one of a number of variables in an INPUT statement, the value for each string variable must be entered on a separate line. In the following INPUT statement,

```
500 INPUT A, B, C$, D$
```

the operator might respond to the INPUT prompt as follows:

```
100, 200, JOHN
MARY
```

The reason for the entry of string data on separate lines is that Atari BASIC allows a comma to be input as part of a string. Therefore, the comma cannot be used as a delimiter. You can test this by entering the following,

```
SMITH, JOHN
```

for one of the string variables in our preceding example.

It is good programming practice to include a prompt message in conjunction with an INPUT statement to let the operator know what data the computer is expecting. This is accomplished by preceding the INPUT statement with a PRINT statement. If the PRINT statement is ended with a semicolon, the prompt message will be displayed on the same line with the INPUT prompt.

```
100 PRINT "ENTER YOUR AGE";  
200 INPUT AGE
```

In the preceding example, the prompt, "ENTER YOUR AGE", will appear on the same line as the INPUT prompt.

Loops

Suppose that you needed to compute the squares 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  
:  
:  
:
```

However, this method is very cumbersome. This 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 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.

Nested Loops

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

```
100 P = 1000
200 FOR Y = 1 TO 10
300 FOR Q = 1 TO 4
400 P = P + P * .02
500 NEXT Q
600 NEXT Y
650 PRINT P
700 END
```

Our preceding example is used to calculate the value of 1000 after 10 years with an interest rate of 8% compounded quarterly.

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 Atari BASIC interpreter cannot match every NEXT statement with a preceding FOR statement, an error will result.

Conditional Statements

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

IF *expression* THEN *statement* ELSE *statement*

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

In the following example, if X is equal to 0, then Y will be set to 1. If X is not equal to 0, Y will be set to 0.

100 IF X = 0 THEN Y = 1 ELSE Y = 0

The IF, THEN, ELSE statement may be shortened to just IF, THEN as shown below.

050 Y = 0
100 IF X = 1 THEN Y = 1

In this example, if X is equal to 1, the statement following THEN will be executed. If X is not equal to 1, program execution will continue with the next program statement (in our example--line 200).

Branching Statements

Branching statements change the execution pattern of programs from their usual line by line execution in ascending line number order. 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
```

would branch program control at line 500 to line 999.

Branching statements are often used in conjunction 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 the IF statement. This is shown in the following example.

```
050 DIM A$(99)
100 PRINT "ENTER THE AMOUNT"
150 INPUT A
200 IF A = 0 THEN GOTO 900
900 PRINT "ARE YOU FINISHED (Y/N)": INPUT A$
910 IF A$ = "N" THEN 100
999 END
```

In our preceding example, if the value input for A has a zero value, then the program will branch to line 900 where the operator will be prompted whether he has finished entering data. In line 910, the program will set up a condition where if the input was 'N', the program will branch to line 100. If the entry was not equal to 'N', the program will continue to line 999.

Note in line 910 that a GOTO statement is not used to precede the line number being branched to. When a line number is indicated following a THEN or ELSE statement, the computer does not require the presence of GOTO, which is assumed.

ON, GOTO Statement

The ON, GOTO statement is a combination of a conditional statement and a branching statement. The use of the ON, GOTO statement is illustrated in the following program.

```
100 INPUT A
200 ON A GOTO 250,260
210 GOTO 999
250 PRINT A: GOTO 100
260 PRINT A ^ 2:GOTO 100
999 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; if 3, to the third, 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.

Subroutines & GOSUB Statements

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.

Illustration 3-5. BASIC Program With a Subroutine

```

050 DIM ER$(50), B$(50)
100 PRINT "ENTER CHECK AMOUNT"
200 INPUT A
300 GOSUB 900
400 PRINT "ENTER PAYEE'S NAME"
500 INPUT B$
600 PRINT B$, A
700 GOTO 100
Subroutine { 900 REM ERROR SUBROUTINE
              910 ER$ = "NOT ALLOWED"
              920 IF A < 0 THEN GOTO 100
              930 IF A > 1000 THEN PRINT ER$
              940 IF A = 0 THEN 999
              950 RETURN
              999 END

```

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.

ON, GOSUB Statement

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

Break Key and CONT

Generally, Atari BASIC programs can be stopped by pressing the Break key. When the Break key is used to stop program execution, a message similar to the following will be displayed.

```
STOPPED AT LINE XXX
```

In actual practice, the XXX will be replaced by the line number where program execution stopped.

Once program execution has been stopped by pressing the Break key, the computer will return to the immediate mode. If you wish program execution to resume, enter the CONT command at the keyboard. Program control will resume with the line following the one where the program break occurred.

System Reset Key

Program execution can also be stopped at any time by pressing the System Reset key. However, System Reset functions somewhat differently than Break.

When System Reset is pressed, the program will stop executing, the display screen will be erased, and the ATARI will return to the immediate mode. You may be able to resume program execution by entering CONT. However, this is not assured. With complex programs, chances are slim that program execution can be resumed once System Reset has been pressed.

STOP STATEMENT

The STOP statement functions in much the same manner as pressing the Break key. The following is an example of a program line containing a STOP statement.

```
500 STOP
```

When the statement is executed, program execution will be halted, and the following message will be displayed.

```
STOPPED AT LINE 500
```

The program will return to the immediate mode, where execution can be resumed by entering CONT.

END Statement

The END statement also causes program execution to halt. An example of an END statement is given below.

```
999 END
```

When an END statement is executed, program execution will halt, the message READY will be displayed on the screen, and the computer will return to the immediate mode.

Execution can be resumed with the line following the END statement by entering CONT.

Unlike the STOP statement, the END statement closes any open input/output channels, sets the screen to graphics mode 0, and turns off all sound voices.

When the Atari runs out of BASIC program statements, an END statement is automatically executed.

Atari BASIC Functions

Functions are used in Atari BASIC to perform predefined calculations or operations on their arguments. All functions use the following format.

function (argument)

function is the keyword for the function. *argument* is a variable, constant, or expression which is to be used in the operation defined by the function.

The following statement is an example of the use of the SQR function.

100 A = SQR(49)

In this example, A would evaluate at 7. SQR is the keyword which describes the square root function. The square root of 49 is, of course, equal to 7.

Functions can be used with arithmetic, relational, and Boolean expressions, as shown in the following statement.

100 X = 100 - 7 * SQR(49)

In an expression containing functions as well as arithmetic, relational, and/or Boolean operators, the function's value is calculated first. In our preceding example, the square root of 49 would be calculated, that value would be multiplied by 7, and the product subtracted from 100.

The various Atari BASIC functions are described in Chapter 5.

CHAPTER 4. ADVANCED ATARI BASIC

Introduction

In this chapter, we will expand on the concepts of BASIC programming that were introduced in Chapter 3. The following topics will be covered.

- String Handling
- Variable Storage
- PEEK
- POKE
- Screen Output Programming
- Input Programming
- Prompt Messages
- INPUT Response Checks
- CHR\$
- ASC
- TAB

Atari ASCII

The Atari can not store characters; it can only store numbers. Before characters can be stored, they must be converted to numbers. Computers use special numeric codes to store characters. Most microcomputers use a code known as ASCII (American Standard Code for Information Interchange).

The Atari uses a special version of ASCII known as Atari ASCII. When we refer to ASCII in this book, we will be referring to Atari ASCII. The Atari ASCII code set is outlined in Appendix C.

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. Atari BASIC allows for all of these.

Substrings

Atari BASIC allows the programmer to extract a portion of a string, known as a **substring**. However, Atari BASIC accomplishes this extraction in a manner which is very different from other versions of BASIC, which use MID\$, RIGHT\$, and LEFT\$ to accomplish this task.

Atari BASIC uses the following configuration to extract a substring.

NAME\$ (*first*, *last*)

Where NAME\$ is the name of the string from which the substring is to be extracted, *first* is the position of the first character from NAME\$ to be included in the substring, and *last* is the position of the last character from NAME\$ to be included in the substring.

For example, if X\$ consisted of the following,

"JOHN JOHNSON"

the substring defined by X\$ (1,4) would consist of "JOHN", and X\$ (6,12) would consist of "JOHNSON". Notice that the blank space in X\$ is counted as one character position.

The first and last character position in a substring specification can be specified with a variable or an expression as well as a constant. Also, the last character position need not be specified. If it is not, the entire right hand portion of the string will be returned beginning with the specified first character.

Substrings can be used to replace characters in larger strings. In

the following program, a substring is used to change X\$ from "JOHN JOHNSON" to "JOHN JACKSON".

```

100 DIM X$(15)
200 X$ = "JOHN JOHNSON"
300 X$(6,12) = "JACKSON"
400 PRINT X$
500 END
RUN
JOHN JACKSON

```

If an error occurs with a substring specification, error number 5 will be displayed.

String Concatenation

The process of joining together one or more strings is known as concatenation. The LEN function is used in conjunction with substrings in concatenation. The LEN function is used to return the length of its string argument. LEN uses the following configuration.

LEN (*string*)

The following program illustrates string concatenation in Atari BASIC.

```

100 DIM X$(15), Y$ (15)
150 X$ = "":Y$ = ""
200 X$ = "JOHN"
300 Y$ = "JOHNSON"
400 X$(LEN(X$) + 1) = Y$
500 PRINT X$
600 END
RUN
JOHNJOHNSON

```

The actual concatenation takes place in line 400. Here, Y\$ is added onto the end of X\$ to form a new X\$. Notice that 1 was added to the result of LEN(X\$). This causes Y\$ to be added beginning at the first blank space following the end of the original X\$.

If line 200 was revised as follows,

```
200 X$ = "JOHN "
```

the following could be output:

```
JOHN JOHNSON
```

The addition of a blank space in X\$ results in one additional blank space being output.

CHR\$ & ASC Functions

As mentioned earlier, characters are represented with the Atari as ASCII codes. Atari BASIC's CHR\$ function can be used to translate an ASCII code to its equivalent character. The following short program illustrates the use of the CHR\$ function.

```
100 PRINT CHR$ (54)
200 PRINT CHR$ (55)
300 END
RUN
6
7
```

The CHR\$ function is often used to represent characters in a statement, when that character can not be represented in its text form. For example, in the following program,

```
100 PRINT CHR$(34); "JOHN JOHNSON"; CHR$(34)
200 END
RUN
"JOHN JOHNSON"
```

quotation marks are specified in the PRINT statement using their ASCII code and the CHR\$ function.

The ASC function returns the ASCII code equivalent for its string argument. If this string is longer than one character, the ASC function returns the ASCII code for just the first character in the string.

The following program illustrates the use of the ASC function:

```
050 DIM A$(20)
100 A$ = "JOHN JOHNSON"
200 PRINT ASC(A$)
300 END
RUN
74
```

Escape Sequences in Strings

Generally, the cursor movement characters may not be included within a string. They may, however, be included if they are preceded by the operator pressing the Escape key.

When the Escape key prefixes a cursor movement key, the combination is known as an **escape sequence**.

The following program will illustrate the use of an escape sequence.

```
100 PRINT "JOHN←N→JOHNSON"
200 END
RUN
JOHN JOHNSON
```

In our example, the symbol ← denotes pressing ESC followed by CTRL-+. The symbol → denotes pressing ESC followed by CTRL-*.

In our previous example, the cursor movement itself was accomplished by using an escape sequence. Each cursor movement is also associated with a character as shown in Table 4-1. By pressing the Escape key twice before the cursor movement key sequence, this character will be output. This is shown in the following program.

```
100 PRINT " ^E↑^E↑^E↑ "
200 END
RUN
↑↑↑
```

In this example, E_E represents pressing the Escape key twice, and \uparrow represents pressing Escape Ctrl--.

The various escape sequences are given in Table 4-1.

Table 4-1. Escape Sequences






























Keyboard Entry	ASCII Code	Echoed Character	String Character
ESC/ESC	27		Escape Code
ESC/CTRL--	28		Cursor Up
ESC/CTRL-=	29		Cursor Down
ESC/CTRL-*	30		Cursor Right
ESC/CTRL-+	31		Cursor Left
ESC/CTRL-<	125		Clear Screen
ESC/SHIFT-<	125		Clear Screen
ESC/BACK S	126		Cursor left, replace with blank space
ESC/TAB	127		Cursor right to next tab stop
ESC/SHIFT-BACK S	156		Delete Line
ESC/SHIFT->	157		Insert Line
ESC/CTRL-TAB	158		Clear Tab Stop
ESC/SHIFT-TAB	159		Set Tab Stop
ESC/CTRL-2	253		Sound Built-in Speaker
ESC/CTRL-BACK S	254		Delete Character
ESC/CTRL->	255		Insert Character

Graphics Characters in Strings

The Atari has 29 graphic characters. These are output by using the Control key in combination with another key. Table 4-2 contains a list of the graphics characters.

The graphics characters can be included in a string with a PRINT statement to output graphics to the screen. For example, the following program,

Table 4-2. Atari Graphics Characters

Decimal Code	ASCII Character	Keystrokes	Decimal Code	ASCII Character	Keystrokes
0		CTRL-,	15		CTRL-O
1		CTRL-A	16		CTRL-P
2		CTRL-B	17		CTRL-Q
3		CTRL-C	18		CTRL-R
4		CTRL-D	19		CTRL-S
5		CTRL-E	20		CTRL-T
6		CTRL-F	21		CTRL-U
7		CTRL-G	22		CTRL-V
8		CTRL-H	23		CTRL-W
9		CTRL-I	24		CTRL-X
10		CTRL-J	25		CTRL-Y
11		CTRL-K	26		CTRL-Z
12		CTRL-L	96		CTRL-.
13		CTRL-M	123		CTRL-;
14		CTRL-N			

```

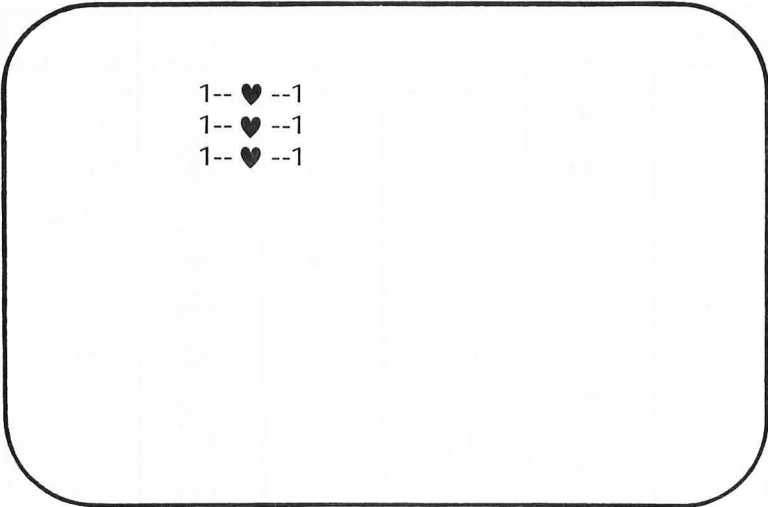
100 DIM A$(20)
200 A$ = "1-- ♥ --1"*
300 PRINT A$:PRINT A$:PRINT A$
400 END

```

would result in a display similar to that shown in Illustration 4-1 when it is run.

* ♥ --is generated by pressing Ctrl-,

Illustration 4-1. Graphics Example Program Output



```
1-- ♥ --1
1-- ♥ --1
1-- ♥ --1
```

Variable Storage

Atari BASIC keeps a list of the variable names used in a program in its **variable name table**. A maximum of 128 variable names can be stored in the variable name table. Therefore, an Atari BASIC program is effectively limited to a maximum of 128 variables. These include numeric, string, and array variables. An array variable name counts as only 1 name in the variable name table, regardless of the number of elements within that array.

Every time a new variable is entered in the immediate mode, that name is added to the variable name table. In the program mode, variables are added to the variable name table as they are encountered in the program.

Variable names are stored in the variable name table until a NEW command is issued. NEW causes the entire variable name table to be cleared.

When a program is saved on cassette with the CSAVE statement, the variable name table is saved on tape along with the program itself. If the program is later loaded back into memory with the CLOAD statement, the variable name table saved on tape will be read into memory and will take the place of the existing variable name table.

PEEK & POKE

The PEEK and POKE statements allow direct access to the Atari's RAM. The Atari can include as many as 65,536 individual addressable RAM memory locations. Each location is assigned a number sequentially as its address, from 0 to 65,536.

Every memory location can store a number in the range 0 through 255. As mentioned earlier, all data to be stored in memory must be converted to a number in this range. The Atari uses various coding strategies for converting BASIC keywords, text data, numeric data, graphics displays, and machine language into a form that can be stored in memory. The Atari knows how to translate the contents in memory (numbers ranging from 0 to 255) by the context in which that data is used.

The PEEK function allows the user to examine the value stored in the memory location named as its argument. For example, in the following statement,

```
100 N = PEEK(1000)
```

the value stored at memory location 1000 will be assigned to the variable N.

The POKE statement is used to place a value in a specified memory location. POKE uses the following configuration,

POKE *address, value*

where the *value* specified is placed in the location given in *address*. *value* and *address* can either be constants or variables. For example, in the following statement,

100 POKE 2000, X

the value stored in variable X will be POKE'd into memory location 2000.

The POKE statement cannot be used to change ROM. ROM is by definition read-only memory, and cannot be altered with the POKE statement.

Screen Output Programming

The PRINT statement is used to display data on the screen. PRINT statement output begins at the cursor's location. Therefore, cursor positioning is the primary factor in sending output to the screen.

As characters are output to the screen, the cursor position is affected. Generally, the cursor moves one column to the right after it has displayed a character. However, if a PRINT statement ends with a carriage return, the cursor will move to the beginning of the next display line. Also, escape sequences can be used to move the cursor in a direction other than towards the right hand side of the screen. Finally, the POSITION statement can be used to move the cursor to any point on the screen. We will cover each of these methods of cursor positioning as well as other concepts of screen output programming in the next few sections.

Using the Carriage Return in Cursor Positioning

The carriage return is generated by pressing the Atari's Return key. The Return key generates the ASCII **end-of-line** (EOL) character. This character causes the cursor to advance to the beginning of the next display line. The EOL character can also be generated by using the CHR\$ function with 155 (the ASCII code

for EOL).

Tab Function

Tabbing on the Atari is very similar to tabbing on a normal typewriter. Tabs are preset along the entire length of a logical line. The first tab position is the left margin (column 2), followed by columns 7, 15, 23, and every eighth column to the end of the logical line.

Tabs work much like commas do when they are used as formatting characters in PRINT statements. However, tabs and commas function completely separately. The column positions set up by commas have no effect on the tab positions, and vice versa.

In the immediate mode, the tab key is used to move the cursor to the next tab position. When the tab key is pressed, the cursor will move to the next tab position without any of the characters it passes over being erased. If the tab key is pressed with the cursor at the last tab stop, the cursor will move to the start of the next logical line.

In the program mode, the cursor is tabbed by using the ASCII code for tab, 127. This can either be accomplished by using the CHR\$ function or by using ESC/TAB within a string.

In addition to the pre-defined tab stops already mentioned, more tab stops can be set in any column desired. In the immediate mode, a tab stop can be set by moving to the desired column and pressing the SHIFT-TAB keys.

Tab stops can also be set with the PRINT statement. The PRINT statement must display a string which causes the cursor to move to the desired position. The tab set character, CHR\$(159) or ESC/SHIFT-TAB, must then occur in the string. For example, in the following statement,

```
100 PRINT "JOHN"; CHR$(159)
```

a tab stop is set in the fourth column.

A tab stop can be cleared in the immediate mode by moving the cursor to the position desired and then pressing CTRL-TAB. In the program mode, a tab stop can be cleared by moving to the desired column and displaying ASCII 158. This code can be displayed either with the CHR\$ function or with ESC/CTRL-TAB.

One final point to keep in mind about tab stops is that whenever a character is output in the space immediately preceding a tab stop, that tab stop no longer has any effect.

Moving the Cursor with Escape Sequences

As mentioned earlier in this chapter, the cursor can be moved by using the escape sequences for cursor control key sequence within a PRINT statement string. For example, in the following statement,

```
100 PRINT " → → JOHN JOHNSON"
```

the symbol → represents pressing the following key sequence:

ESC/CTRL-*

This key sequence causes the cursor to move one position to the left each time it is pressed.

Cursor control escape sequences can also be included in a PRINT statement string by using the ASCII code for that sequence with the ASC\$ function. For example, in the following statements,

```
100 DIM A$(10)
200 A$ = CHR$(29)
300 PRINT A$:PRINT A$:PRINT A$
```

the string variable A\$ is set to the ASCII code for cursor down. In line 300, the three PRINT statements cause the cursor to be moved down 3 lines.

These cursor control sequences do not erase any of the characters that they pass over.

Home Cursor

The **home** position can be defined as the upper left-hand corner of the video display. The home cursor control sequence moves the cursor to the position and erases all existing data on the screen as well.

Home cursor is frequently used to position the cursor and erase the screen in Atari BASIC. Home cursor can either be accomplished by using the ASCII code for home cursor, 125, with the CHR\$ function, or by using either of the following escape sequences:

ESC/CTRL-<
ESC/SHIFT-<

with the PRINT statements.

POSITION Statement

The POSITION statement can be used to place the cursor at any location on the screen. The POSITION statement is used with the following configuration,

POSITION *column, row*

where *column* is the number of the column to be moved to, and *row* is the number of the row to be moved to.

In actuality, the POSITION statement does not cause the cursor to be moved. POSITION merely changes the values in the Atari's memory where the cursor location is stored. When data is subsequently displayed on the screen, that data will be displayed at these new display coordinates.

The display row number is stored in memory address 84, and the column number is displayed in address 85. The contents of these locations can be examined with the PEEK function. For example, the following statements,

PEEK (84)
PEEK (85)

will return the row and column numbers respectively.

When PRINT is used to output data to the screen, the previous cursor position is stored in memory. Memory address 90 contains the last row number, and memory address 91 contains the last column number. Again, the PEEK function can be used to examine the contents of these memory addresses.

Remember, rows are numbered from 0 to 23, and columns are numbered from 0 to 39.

Changing the Display Screen Margins

The standard left margin on the display screen is column 2. The standard right margin is column 39. The Atari uses memory address 82 to store the column number of the left margin, and location 83 to store the column number of the right margin.

The POKE statement can be used to change either the left or right margins. The following statements would reset the left margin to column 5, and the right margin to column 30.

```
POKE 82, 5  
POKE 83, 30
```

Screen Input Programming

Input programming is a vital part of BASIC programming. Nearly every BASIC program requires some form of operator input. In the following few sections, we will discuss programming practices that are designed to make operator input efficient and as error-free as possible.

Prompt Messages

One programming principle that should nearly always be followed in input programming is to include a prompt message with the INPUT statement. An example is given below.

```
100 PRINT "ENTER YOUR AGE";  
200 INPUT AGE
```

In general, it is advisable to keep prompt messages as brief as possible--as long as the message is clear to the user. Avoid prompt messages which are overly wordy.

When long prompt messages are being used, it is a good practice to place the prompt message on one line, and the input response on the next line. For example, the following statement,

```
100 PRINT "ENTER OPERATION CODE (1 = ADD; 2 = DEL)"
200 INPUT X
```

would result in the following display:



```
ENTER OPERATION CODE (1 = ADD; 2 = DEL)
?
```

Input Response Checks

A well-designed program should check the user's response to an Input statement to be certain that no obvious input errors have been made. If such an error was made, the program should detect the error and force the user to re-enter the data.

Examples of input errors that can occur are numeric entries that are outside of the allowed range, string entries that are longer than allowed for by the Input statement's variable, and an input

response other than that prompted for.

The very nature of the Input statement prevents certain errors from occurring as these are detected by the BASIC interpreter. For example, if a numeric entry is made when a string variable is specified with the Input statement, an error will occur. Likewise, if a string entry is made when a numeric variable is specified with the Input statement, an error will occur.

However, many Input entry errors will not be detected by the BASIC interpreter. Serious errors can occur when the wrong data is entered in response to an Input statement. It is a good programming practice to check the operator's response to an Input statement. This can either be accomplished with one or more IF-THEN statements, or with ON-GOTO or ON-GOSUB statements.

For example, in the following program, the operator's input is checked with two IF-THEN statements. If the response is neither of the following,

Y, N, y, n

the program will branch back to line 1200 for a new entry.

```
1000 DIM A$(20)
1100 PRINT
1200 PRINT "Enter Your Response (Y/N)"
1300 INPUT A$
1400 A$ = A$(1,1)
1500 IF A$ = "Y" OR A$ = "y" THEN 1800
1600 IF A$ = "N" OR A$ = "n" THEN 9999
1700 GOTO 1300
1800 REM Subroutine For 'Yes' Response
1900 PRINT "YES"
9999 END
```

CHAPTER 5.

ATARI BASIC REFERENCE GUIDE

This chapter provides descriptions and examples of the correct syntax for Atari BASIC.

Each of the reserved words are listed in alphabetical order, along with an appropriate abbreviation, if applicable.

The following notation will be used to describe the configuration of each of the commands, statements, or functions.

1. Capitalized words are keywords.
2. Items enclosed in brackets [] are optional.
3. Ellipsis (...) represents repetition.
4. Punctuation (except brackets) must be included as shown.
5. The following symbols will be used:

LN	Line number
EX	Algebraic or logical expression (i.e. $X > 5$, $3 + X$, $X = 7$)
X, Y, Z	Numeric variable name
X\$, Y\$, Z\$	String variable name
a, b, c	Any number or numeric expression
a\$, b\$, c\$	String value

ABS

The ABS function returns the absolute value of its argument.

Configuration

$X = \text{ABS}(a)$

Example

```
PRINT ABS(-81)  
81
```

ADR

The ADR function returns the memory address of the argument. The argument must be a string variable or a string constant.

In BASIC, a machine language program can be put in a string variable. However, the operating system moves variables around to efficiently use memory. As a result, to call a machine language routine, the ADR function is used to locate the string.

Configuration

$X = \text{ADR}(a\$)$

Example

$X = \text{ADR}(B\$)$

AND

AND is used between two expressions, and returns the value 1 if they are both true, and 0 if either one is false.

CONFIGURATION

EX AND EX

The conditions of true and false are represented in the computer by the logical values 1 and 0. As a result, the logical operators (AND, OR, and NOT) generate only the values 1 and 0. The AND operation can be explained by the following truth table.

EX1	EX2	RESULT
1	1	1
1	0	0
0	1	0
0	0	0

AND is generally used in an IF/THEN statement with relational expressions. For example:

```
10 X = 10
20 Y = 30
30 IF X = 10 AND Y > 100 THEN END
40 PRINT "CONDITIONS WERE NOT MET"
RUN
CONDITIONS WERE NOT MET
```

In this example, AND is used in an IF/THEN statement which ends the program if both conditions are true. The first expression of the AND statement is $X = 10$. This is true because X is assigned the value 10 in line 10. The second expression, $Y > 100$, is false because Y is assigned the value 30 in line 20. As a result, EX1 is true and EX2 is false. This corresponds to the truth table where $EX1 = 1$ and $EX2 = 0$. The result from the table is 0 (false), so the condition of the IF/THEN statement is false, and the next line is executed.

The AND operator can also be used with algebraic expressions like $5 * Y$, $3 + X$, $X ^ 2$, etc. However, these must also be converted to logical 0 or 1. The computer does this by assigning the logical value 0 to any expression that equals 0. Any expression that does not equal 0 is assigned the logical value 1. For example, the logical value of $5 * 0$ is 0. The logical values of $3 + 1$, $2 ^ 2$, 3 and $\text{COS}(45)$ are all 1.

Example

```

10 X = 3
20 IF X ^ 2 AND 3 - X THEN END
30 PRINT "X IS EITHER 3 OR 0"
RUN
X IS EITHER 3 OR 0

```

This example uses AND in an IF/THEN statement that ends the program if X squared and 3 - X both are not equal to zero. Since X is assigned the value 3, the first part of the AND statement equals 3 squared. This is a logical 1 because 3 squared is non-zero. However, the second expression, 3 - X, is equal to zero, which is the logical 0. Since EX1=1 and EX2=0, the AND statement is false, and the next statement is executed (line 30).

ASC

The ASC function returns the ASCII code for the first character of a string. The argument of ASC can be a string variable or constant.

CONFIGURATION

$X = \text{ASC}(a\$)$

EXAMPLE

```

10 DIM B$(10)
20 B$ = "ZEBRA"
30 PRINT ASC(B$)
RUN
90

```

ATN

The ATN function returns the arctangent of the argument. The result will be in radians unless degrees are specified.

CONFIGURATION $X = \text{ATN}(a)$ **EXAMPLE**

```
PRINT ATN(.576)  
0.5225854816
```

BYE

BYE switches the system to the Memo Pad mode. The system has no computing ability, and only the keyboard and display are functional. The operator can experiment with the keyboard without affecting the system. The system will return to BASIC when the SYSTEM RESET key is pressed.

The operations of the computer and other devices (disk drive, modem, etc.) are not at all affected by the Memo Pad. For example, if a program is in memory, and a disk and modem are being used, a BYE command will switch to Memo Pad. However, SYSTEM RESET will restore the computer to BASIC, and all other devices will still be ready to operate. The program in memory will be unchanged.

CONFIGURATION

BYE

EXAMPLE

BYE

CLOAD (CLOA.)

The CLOAD command is used to load a previously recorded program into the computer's memory. The program must have been stored on a cassette with a CSAVE or SAVE command.

At the sound of the tone, press PLAY on the program recorder,

then press RETURN on the keyboard. The tape must be correctly positioned before CLOAD is executed.

The CLOAD command clears the memory before the program is loaded from the tape.

CONFIGURATION

CLOAD

EXAMPLE

CLOAD

CHR\$

The CHR\$ function returns the character with the ASCII code specified by the argument. Although argument values can range from 0 to 65535, the ASCII code corresponds to the numbers from 0 to 255.

CONFIGURATION

X\$ = CHR\$(a)

EXAMPLE

PRINT CHR\$(65)
A

CLOG

The CLOG function returns the base 10 logarithm of the argument.

CONFIGURATION

X = CLOG(a)

EXAMPLE

PRINT CLOG(4)
0.6020599914

CLOSE (CL.)

The CLOSE statement closes a channel that has been opened for input, output, or both. However, closing a channel that has not been opened will not cause an error.

The argument of a CLOSE statement must be the same as in the corresponding OPEN statement. A channel that has been opened for the use of a particular I/O device must be closed before it is used for another device.

CONFIGURATION

CLOSE #a

EXAMPLE

CLOSE #3

CLR

The CLR command clears the values of the variables in the memory. However, the variable name table remains unchanged. As a result, the CLR command does not reduce the number of variable names. After using CLR, all strings, arrays, and matrices must be dimensioned again.

CONFIGURATION

CLR

EXAMPLE

CLR

COLOR (C.)

In graphics modes 0 through 2, the COLOR statement is used to choose the character that will be placed on the screen with a PLOT statement.

CONFIGURATION

COLOR a

In all graphics modes, the argument of the COLOR statement must be positive, and if it is not an integer, it will be rounded off.

In mode 0, the text is printed in the same color as the background. Only the luminance of the color can be chosen. For example, if the background is chosen to be green, the text must be green, but it can be any brightness. The COLOR statement indicates the character that is to be printed with the next PLOT statement. In graphics mode 0, the COLOR statement has no effect on the color of the character. Table 9-7 lists the characters that correspond to the COLOR statement in graphics mode 0.

EXAMPLE

```
10 GRAPHICS 0
20 FOR I = 1 TO 5
30 READ X
40 COLOR X
50 PLOT 10 + I, 10
60 NEXT I
70 DATA 65, 84, 65, 82, 73
```

In the previous example, the word ATARI is printed at the center of the display. Each data item is read individually at line 30, and becomes the argument of the COLOR statement in line 40. The loop is repeated 5 times, and each time the COLOR statement has a different value as its argument. It can be seen from Table 9-7 that in graphics mode 0, COLOR 65 indicates the character A.

After the COLOR 65 statement has been executed, any PLOT or

DRAWTO statement will be executed with the character A until another COLOR statement has been executed.

EXAMPLE

```
10 GRAPHICS 0
20 COLOR 65
30 PLOT 0,0
40 DRAWTO 10,10
```

The preceding program would print the character A in the upper left-hand corner of the screen because of the PLOT 0,0 statement. The DRAWTO 10,10 would cause a diagonal line consisting of the character A to appear on the display. A character would appear at the positions (0,0), (1,1), (2,2)...(10,10).

The display looks like white characters on a blue background. Actually, the "white" is very bright blue. The intensity of the characters can be chosen with a SETCOLOR statement.

The COLOR statement has a different function in graphics modes 1 and 2. Modes 1 and 2 have fewer characters available than Mode 0, but each character can be printed in 4 colors.

The difference between modes 1 and 2 is the size of the character. The characters in mode 2 are twice the height of mode 1, but are the same width.

Table 9-4 lists the values of the COLOR statement arguments for each character in 4 colors. The columns of the table correspond to the 4 color registers. The standard character set will be used unless the alternate character set is specified with the statement POKE 756, 226. To return to standard characters, POKE 756, 224.

EXAMPLE

```
10 GRAPHICS 1
20 FOR I = 1 TO 5
30 READ X
40 COLOR X
50 PLOT 6 + I, 0
60 NEXT I
70 DATA 65, 116, 193, 114, 73
```

The previous example displays the word ATARI at the top of the display in three colors. The data is read at line 30 and becomes the argument of the COLOR statement at line 40.

The COLOR statement chooses the character and the color register to be used in the display. From Table 9-4, COLOR 65 indicates the character A in color register 0. COLOR 116 indicates the character T in color register 1.

The color registers are assigned specific information about the color to be used. Color registers can be changed with a SETCOLOR statement, but if no SETCOLOR statement is executed, a standard set of default colors are used. The default colors for graphics mode 1 and 2 are as follows:

COLOR REGISTER	DEFAULT COLOR
0	ORANGE
1	GREEN
2	BLUE
3	RED
4	BLACK

Color register 0-3 can be chosen for any character, but color register 4 is used for the background and border.

In the previous example, the first character displayed was an A in color register 0. Since no SETCOLOR was executed, the A will be orange. The T will be green because COLOR 116 is in color register 1.

If the same program was executed in the alternate character set, by executing POKE 756, 226 after the GRAPHICS statement, the word ATARI would appear in lower case letters. Also, in the alternate character set, a "heart" character will appear in every blank space. This occurs because the standard character set puts a space (COLOR 32) in areas where no character has been assigned. When the conversion to the alternate character set occurs, COLOR 32 is interpreted as a "heart" in color register 0 (Table 9-4). As a result, an orange "heart" will appear in every space except where the word ATARI appears.

In graphics modes 3 through 7, the COLOR statement is used to choose the color register that will be used to plot points and draw lines.

Graphics modes 3 through 7 are different from modes 0 through 2 because modes 0, 1 and 2 are used to place characters on the screen. Graphics modes 3 through 7 are used to place picture elements (pixels) on the screen. A pixel is a rectangle that is referred to by its coordinates (column and row) on the display. In modes 3 through 7, the COLOR statement actually chooses a color register, not a character.

EXAMPLE

```
10 GRAPHICS 3
20 FOR T = 0 TO 3
30 COLOR T
40 PLOT T,0
50 NEXT T
```

The previous example displays the 4 colors of graphics mode 3. Line 40 plots a pixel at column T, row 0. The color of the pixel is determined by the last COLOR statement. The first time through the program, T is set equal to 0 at line 20. Line 30 indicates that color T is used. Since no SETCOLOR statement was executed, the default colors are used.

GRAPHICS MODES 3, 5, and 7	
COLOR NUMBER	DEFAULT COLOR
0	ORANGE
1	GREEN
2	BLUE
3	BLACK

As a result, when $T=0$, the color is orange. The PLOT statement at line 50 colors the pixel at column 0, row 0 orange. The next pixel, at column 1, row 0 is colored green. The pixel at column 2, row 0 is blue and the next one is black.

In graphics modes 4 and 6, the COLOR statement is used in the same fashion as in graphics modes 3, 5, and 7. However, modes 4 and 6 have only two colors, and the default colors are as follows.

GRAPHICS MODES 4 and 6	
COLOR NUMBER	DEFAULT COLOR
0	BLACK
1	ORANGE

Graphics mode 8 has only one color, with two brightness levels. As a result, the COLOR statement is used to select the brightness of a pixel. In other words, COLOR 1 causes the next plotted pixel to be visible. COLOR 0 causes the next plotted pixel to be the same as the background.

In graphics mode 8, the pixels are very small, and the graphics are slow. It sometimes is useful to draw an entire area, then "erase" what is not wanted. This is often faster than drawing only what is wanted. This can be done by drawing an area using COLOR 1, then "erasing" by using COLOR 0.

COM

COM is used interchangeably with DIM in dimensioning strings, arrays, and matrices.

CONFIGURATION

$$\text{COM } \begin{matrix} X(a[b]) \\ X\$(a) \end{matrix} \left[\begin{matrix} Y(c[d,]) \\ Y(c) \end{matrix} \right]$$
EXAMPLE

```
COM B$(50), A(10,10)
```

CONT (CON.)

The CONT command causes a program which had been stopped to continue execution at the next numbered line. A program will be stopped because of an error, SYSTEM RESET, BREAK, END, or STOP.

In any situation, the use of CONT will cause the rest of the current line of code to be ignored. As a result, executing BREAK and CONT during a program may cause serious problems. When a program is stopped using BREAK, there is no way to be sure the program will resume where it was stopped. Important steps may be interrupted or skipped, and loops may be improperly exited.

A program can be continued after an error, but the entire line of the error will be skipped.

A program can be continued after a SYSTEM RESET, but this will generally have negative results. All I/O channels will be closed, the computer will return to the immediate mode, the screen will be cleared, graphics mode 0 will resume, etc.

CONFIGURATION

```
CONT
```

EXAMPLE

```
CONT
```

COS

The COS function returns the cosine of its argument. The argument will be assumed in radians unless a DEG statement precedes the COS statement.

CONFIGURATION

$X = \text{COS}(a)$

EXAMPLE

```
10 DEG
20 X = COS(180)
30 PRINT X
RUN
-1
```

CSAVE (CS.)

The CSAVE command is used to copy the program in the computer's memory on cassette tape. Only CLOAD can be used to read a program that was stored using CSAVE.

When the tape is properly positioned, enter CSAVE. The tone will sound twice as a signal to press the cassette recorder's PLAY and RECORD keys, followed by pressing RETURN on the Atari keyboard.

If channel 7 is open for another device, an error will occur, but the channel will be closed. A repeat of CSAVE will then be successful.

CONFIGURATION

CSAVE

EXAMPLE

CSAVE

DATA (D.)

The DATA statement supplies a list of information that is used in a program through READ statements. A DATA statement can include numeric values, string values, or both. String variables must have been dimensioned before being read.

Data items are separated by commas. Therefore, string values that contain commas will be read as separate data items. For example, DATA DOE, JOHN is a DATA statement with two data items. However, DATA DOE. JOHN has only one item.

CONFIGURATION

$$\text{DATA } \begin{matrix} a \\ a\$ \end{matrix} \begin{bmatrix} b \\ b\$ \end{bmatrix} \dots$$

Data must be read into the correct type of variable. A string variable can accept data in any form.

EXAMPLE

```
10 DIM A$(20)
20 FOR I = 1 TO 5
30 READ A$:? A$
40 NEXT I
50 DATA TOM C., 25,,3 + 4 * %,247
RUN
TOM C.
25

3 + 4 * %
247
```

The preceding example shows correct data for a string variable. Notice the blank line in the output that corresponds to the two commas in a row. This is read as a string value with no characters and length equal to zero.

If only 4 data items had been supplied with this program, the message: ERROR-6 AT LINE 30 would have been displayed to

notify the user that not enough data was supplied.

Numeric variables can only accept numbers as input. Standard notation and scientific notation are both acceptable. For example, 3.14159266, 2.85E-10, .0001, 35 and -45 are all acceptable data items. Expressions will not be evaluated. They will cause an Input Statement Error (#8). Numeric data must not include commas.

EXAMPLE

```
10 DIM A$(10)
20 FOR I = 0 TO 4
30 READ A$, A
40 PRINT A$, A
50 NEXT I
60 DATA PENCILS, 20,PENS,25,RULERS,40,ERASERS,50,
    PAPER,200,GLUE,5
```

The preceding example shows a correct sequence for reading string and numeric data into correct variables. However, the READ statement is only called 5 times, and there are 6 sets of data. This will not cause an error, but the last set of data (GLUE,5) will never be read.

DATA statements can appear anywhere in a program, even after an END statement. However, any statement that follows a DATA statement on the same line will not be executed.

Data can only be read once unless a RESTORE statement is executed. The correct use of RESTORE is also explained in this chapter.

DEG (DE.)

The DEG statement causes the trigonometric functions to be performed in degrees instead of radians. The functions will be performed in radians until degrees are specified. Also, radians will be used after a SYSTEM RESET, NEW, or RUN command.

CONFIGURATION

DEG

EXAMPLE

```

10 DEG
20 PRINT SIN(90)
RUN
1

```

The example shows that the sine of 90° is 1. If the DEG statement was not present, the result would be 0.8939970243.

DIM (DI.)

The DIM statement is used to set aside memory space for strings and 1 or 2 dimensional arrays. Two dimensional arrays, or matrices, can be used to make tables of values.

CONFIGURATION

$$\text{DIM} \quad \begin{matrix} X(a[,b]) \\ X\$(a) \end{matrix} \left[\begin{matrix} Y(c[,d]) \\ Y\$(c) \end{matrix} \right] \dots$$

A DIM statement can include any combination of numeric and string variable dimension statements. For example, DIM A(10,10), B(9), A\$(90), B\$(90) dimensions all four variables in one statement.

A string variable can contain only one string. The dimension of a string variable indicates the maximum number of characters that the string variable can contain.

EXAMPLE

```
10 DIM A$(10)
20 READ A$
30 PRINT A$
40 DATA INDEPENDENCE DAY
RUN
INDEPENDEN
```

The preceding example shows that the string variable A\$ is dimensioned to 10 characters at line 10. However, during the program, A\$ is assigned a 16 character string with the READ statement at line 20. Since room for only 10 characters was set aside in memory, only the first 10 characters of the DATA item are assigned to A\$. The PRINT statement in line 30 displays the contents of A\$. It can be seen from the output that A\$ only has 10 characters.

The DIM statement must be executed before an INPUT or READ occurs. If the DIM statement of the previous example was deleted, the following message would occur.

ERROR-9 AT LINE 20

If a variable is dimensioned twice in the same program (without CLR), ERROR-9 occurs.

The maximum size of string variables depends on the amount of available memory at the time of the DIM statement.

Dimensioning numeric variables determines the number of elements that the variable can contain, not the length. A subscript is the number that follows a variable name (in parentheses) and indicates which element of that variable is considered. The following example shows how to assign 4 values to a subscripted variable.

EXAMPLE

```

10 DIM X(3)
20 FOR I = 0 TO 3
30 READ X:X(I) = X
40 PRINT X(I),
50 NEXT I
60 DATA 12, 14, 13, 15
RUN
12      14      13      15

```

Notice that 4 values can be assigned to a variable that has a dimension of 3. This is possible because each array's initial element has a subscript of 0. The array can be represented as a table of values as shown in the following illustration.

	0	1	2	3
X	12	14	13	15

The number in the DIM statement indicates the largest subscript that can be used.

It should be noted from the example (line 30) that subscripted variables cannot be used in a READ statement. As a result, a separate statement is needed to assign the subscripted variable. The assignment statement can be on the same line (as shown here) or on a separate line.

Numeric variables can also be used with two subscripts. This results in a two dimensional array, or matrix. For example, if X is dimensioned in the statement DIM X(3,2) the following table would result.

	0	1	2
0			
1			
2			
3			

DOS (DO.)

CONFIGURATION

DOS

EXAMPLE

DOS

The DOS command is used to display the DOS utilities Menu. DOS must be present if the DOS command is to be used. If DOS is not present, the system will be put into the Memo Pad mode. To return to BASIC from Memo Pad, press SYSTEM RESET.

When the DOS command is executed, all I/O channels will be closed except channel 0. The display is cleared and the sound voices are shut off. Also, the color registers resume their default values.

The Disk Operating System Menu is a list of 15 disk functions. There are two versions of the Disk Operating System, version 1.0 and version 2.0S. The DOS command has a different effect in each of the two versions.

In version 1.0, the DOS Menu appears on the display as soon as DOS is executed.

DISK OPERATING SYSTEM 9/24/79
COPYRIGHT 1979 ATARI

- | | |
|-------------------|-------------------|
| A. DISK DIRECTORY | I. FORMAT DISK |
| B. RUN CARTRIDGE | J. DUPLICATE DISK |
| C. COPY FILE | K. BINARY SAVE |
| D. DELETE FILE(S) | L. BINARY LOAD |
| E. RENAME FILE | M. RUN AT ADDRESS |
| F. LOCK FILE | N. DEFINE DEVICE |
| G. UNLOCK FILE | O. DUPLICATE FILE |
| H. WRITE DOS FILE | |

A program that is in memory will not be affected by a DOS statement in version 1.0. However, disk operations J or O will erase the contents of the memory. For example, if a program is in memory, and a DOS command is executed, followed by DUPLICATE DISK or DUPLICATE FILE, the program will be gone when the system returns to BASIC.

DISK OPERATING SYSTEM II VERSION 2.0S
COPYRIGHT 1980 ATARI

- | | |
|--------------------|-------------------|
| A. DISK DIRECTORY | I. FORMAT DISK |
| B. RUN CARTRIDGE | J. DUPLICATE DISK |
| C. COPY FILE | K. BINARY SAVE |
| D. DELETE FILE(S) | L. BINARY LOAD |
| E. RENAME FILE | M. RUN AT ADDRESS |
| F. LOCK FILE | N. CREATE MEM.SAV |
| G. UNLOCK FILE | O. DUPLICATE FILE |
| H. WRITE DOS FILES | |

In DOS 2.0, DOS consists of 2 files, DOS.SYS and DUP.SYS. DUP.SYS must be present on the diskette in drive 1 or the Atari will return to BASIC. DUP.SYS was a portion of memory where BASIC programs normally reside. In order to save any BASIC program residing in this area of memory, the Atari will save that program onto the MEM.SAV file on drive 1--if that file exists.

Once these operations have been completed, the DOS utilities menu will appear. You can return to BASIC by choosing menu item B or by pressing the System Reset key.

DRAWTO (DR.)

The DRAWTO statement is used in the graphics modes to draw a line. The arguments of the DRAWTO statement indicate the column and row where that line ends.

CONFIGURATION

DRAWTO a,b

Both arguments of a DRAWTO statement must be positive, and if they are not integers, they will be rounded off. The arguments must also lie within the range of the display. For example, GRAPHICS 3 has 40 columns and 20 rows. DRAWTO 40,20 would result in ERROR-141. Since the columns are numbered 0 to 39 and the rows are numbered 0 to 19, DRAWTO 40, 20 contains arguments which lie outside the range of display.

A DRAWTO statement must occur after a PLOT statement. PLOT determines the starting point of the line, and DRAWTO determines the end point. A DRAWTO statement can follow another DRAWTO statement, if the first DRAWTO is preceded by a PLOT statement.

EXAMPLE

```
10 GRAPHICS 3
20 COLOR 1
30 PLOT 5,5
40 DRAWTO 10,5
50 DRAWTO 10,10
60 DRAWTO 5,10
70 DRAWTO 5,5
```

A DRAWTO statement that follows another DRAWTO statement will use the end of the last line to start the new line. The previous example began by plotting a point at line 30, then proceeded to

draw the 4 sides of a square in Lines 40, 50, 60, and 70.

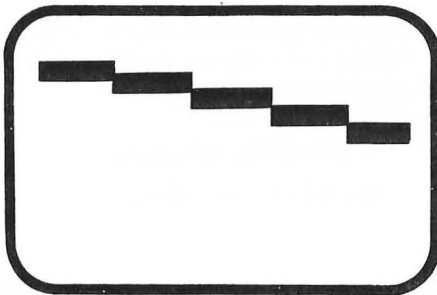
The DRAWTO statement can also be used in graphics modes 0, 1, and 2. However, the PLOT statement in the text modes (0, 1 and 2) places a character on the display. The COLOR statement determines the character that is printed (Tables 9-7 and 9-4). As a result, the DRAWTO statement in the text mode creates a line of characters.

EXAMPLE

```
10 GRAPHICS 2
20 COLOR 65
30 PLOT 0,0
40 DRAWTO 9,9
```

The previous example specifies graphics mode 2 in line 10. Line 20 indicates the character that appears on the display (Table 9-4). The PLOT statement in line 30 places an orange, uppercase A at column 0, row 0. The DRAWTO statement makes a diagonal line, consisting of the character A. The characters appear at the positions (0,0), (1,1),(2,2),... (9,9).

The line drawn with a DRAWTO statement is either composed of picture elements or characters. When a diagonal line is drawn using PLOT and DRAWTO, the line appears in steps. This occurs because the line is drawn with characters or picture elements that are relatively large.



A "line" drawn with PLOT and DRAWTO.

END

An END statement ends the execution of the program. An END is not necessary at the end of a program because execution stops automatically after the last line of code. However, it is good programming technique to end BASIC programs with an END statement.

CONFIGURATION

END

When an END statement is executed, all I/O channels will be closed except 0, the display will be set to graphics mode 0, and all sound will be turned off.

EXAMPLE

```
10 INPUT X
20 IF X <= 10 THEN END
30 PRINT "X IS LARGER THAN 10"
40 GOTO 10
```

The previous example will end only if a value of X is entered which is less than or equal to 10.

ENTER

ENTER is used to recover programs that have been saved on a cassette or disk. ENTER can only be used to load programs that were saved with the LIST statement.

CONFIGURATION

ENTER device[:filespec]

When an ENTER statement is executed, the computer's memory is not erased. As a result, the new program being loaded will be put into memory together with any existing program lines. For

example, if the program in memory contained line numbers 10, 20, 30..., and the program being loaded (using ENTER) contained line numbers 5, 15, 25, 35,..., the resulting program in RAM would include the line numbers from each of the two programs.

ENTER does not alter the program in memory unless the program being entered has the same line numbers as the program being loaded. For example, if the program in memory contains line numbers 10, 20, 30, 40, 50, and 60 and the program being entered contains 10, 20, 30, 45, 55, 70, 80, and 100, the new program in memory will contain all of the newly entered program, but only lines 40, 50, and 60 of the original program. The original lines 10, 20, and 30 in RAM will be replaced with lines 10, 20, and 30 being loaded from cassette or disk. Lines 40, 50, and 60 of the original program remain unchanged.

ENTER is the only Atari BASIC statement that can recover a program without clearing the memory first.

When ENTER is used with the program recorder, the tape must be in the correct position prior to execution. When the ENTER statement is executed, the tone will sound once to remind the operator to press PLAY on the recorder. The recorder will be activated after the RETURN key on the keyboard has been pressed.

When ENTER is used with a disk, the DOS must have been booted first. If more than one disk is being used, the number of the disk must be specified.

EXAMPLES

ENTER "C"

ENTER "D2:JONES"

EXP

The EXP function returns the exponential of the argument. The exponential is the value of $e(2.71828179\dots)$ raised to the power of the argument.

CONFIGURATION

$X = \text{EXP}(a)$

EXAMPLE

```
PRINT EXP(5)
148.413155
```

FOR (F.)

A FOR statement is used with a NEXT statement to form a repetitive loop within a program.

CONFIGURATION

FOR A = a TO b [STEP c]

Every FOR statement must have a corresponding NEXT statement.

EXAMPLE

```
10 FOR I = 1 TO 5
20 PRINT I;
30 NEXT I
RUN
12345
```

In the previous example, the FOR/NEXT loop is repeated five times. Line 20 is the only statement inside the loop, however, any number of program lines can be placed within a loop.

In line 10, I is assigned the value 1. I is referred to as a counter. The value of I is incremented where a NEXT I statement is executed. Here, the program returns to the FOR statement, where I is incremented by one. This loop is repeated until I is set equal to 5. When the counter (I) has been set equal to the value (5), the loop has been executed, the program will proceed with the statement following NEXT I.

A FOR/NEXT loop can use a STEP statement to increment the counter by a value other than 1.

EXAMPLE

```
10 FOR J = 1 TO 2 STEP .5
20 PRINT J,
30 NEXT J
RUN
1      1.5      2
```

The preceding example contains a FOR/NEXT loop which increments the value of J by .5 each time the loop is executed.

A FOR/NEXT loop can also be used to decrease the value of the counter. This can be accomplished by using the optional STEP statement within the FOR statement. If the STEP statement has a negative argument, the counter is decreased each time the loop is executed. The following example illustrates a FOR/NEXT loop where the counter is decremented rather than incremented.

EXAMPLE

```
10 FOR K = 10 TO 5 STEP -2
20 PRINT K,
30 NEXT K
RUN
10      8      6
```

This loop begins at line 10 by assigning the counter (K) the value 10. At line 20 the value of K is printed. When line 30 is encountered, execution continues at line 10, because the NEXT statement returns the program to the preceding FOR statement. The value of the counter is changed by the argument of STEP. Since the STEP value is -2, the counter is decreased by 2. The value of the counter is changed to 8. At line 20, the new value of K is printed. Line 30 is executed again, so the program returns to the FOR statement at line 10. The counter is again decremented by 2. The new value of K is 6. At line 20, this K value is printed.

When line 30 is executed again, the program does not return to

line 10. The current value of the counter is 6, and if the counter was to be decremented again, the counter would be 4. However, 4 is less than the final value which is specified in the FOR statement (the argument of TO). As a result, the loop does not continue after K = 6 because another decrement would make the counter less than the final value (5).

If the counter of a loop is being incremented, the loop will be executed until the counter would exceed the final value if it were incremented again. For example: FOR J = 1 TO 4 STEP 2 would be executed with J equal to 1 and 3. The counter (J) would exceed the final value (4) if it were incremented again.

A FOR/NEXT loop should be executed as if it were a single statement. An attempt to branch into a FOR/NEXT loop will cause an error.

EXAMPLE

```
10 GOTO 30
20 FOR I = 1 TO 10
30 PRINT I
40 NEXT I
RUN
ERROR- 13 AT LINE 40
```

In general, branching out of a FOR/NEXT loop will not cause an error. However, exiting a loop before it has completed should be avoided.

FRE

The FRE function returns the number of bytes of memory available. The FRE function requires an argument, but that argument has no effect on the value returned.

CONFIGURATION

X = FRE (a)

EXAMPLE

PRINT FRE(0)

GET (GE.)

The GET function reads 1 byte from a channel that has been opened for input. GET is used with the keyboard, display, Program Recorder, or disk.

CONFIGURATION

GET #a, X

The first argument of a GET statement indicates the I/O channel that will be used. If the first argument is not an integer, it is rounded off. The second argument names the variable that will be assigned the value read from the channel. This value will be an integer between 0 and 255.

For example, if data is being accepted from the Program Recorder, the GET statement must be preceded by an OPEN statement. The OPEN statement must include the number of the I/O channel, the device name, and an input operation code. Numbers that are not integers are rounded off.

EXAMPLE

```
10 OPEN #3, 4, 0, "C"  
20 FOR J = 1 TO 100  
30 GET #3, X  
40 PRINT CHR$(X)  
50 NEXT J  
60 CLOSE #3
```

The previous example shows the correct format for using a GET statement. Line 10 opens the I/O channel and specifies channel #3 for input with the Program Recorder. The channel number can be any number from 1 through 7, but the channel must not be open for another device. The second argument of the OPEN statement (4) indicates that the device will be used for input.

Line 20 is the first line of a FOR/NEXT loop. The loop ends with the NEXT statement at line 50. The initial value of the counter (J)

is 1, and the final value is 100. The counter is incremented by 1 each time the loop is executed, so the loop will be executed 100 times. Lines 30 and 40 both appear inside the loop (between FOR and NEXT). As a result, lines 30 and 40 are repeated 100 times. Each time line 30 is executed, an integer between 0 and 255 is assigned to the variable X. Line 40 prints the character that has the ASCII code specified by X. Line 60 closes the I/O channel.

GET is used with the disk in the same fashion as it is used with the Program Recorder. However the OPEN statement must include a file specification. The first argument of the OPEN statement is a channel number. Any channel from 1 to 7 can be used if it is not already open. The second argument is the operation being performed. GET can be used with the disk if the OPEN statement has a second argument of 4 (input) or 12 (input and output). For example, OPEN #2, 12, 0,"D:BUDGET" is a correct OPEN statement for using GET with a disk. GET assigns the next byte read from the disk to the variable specified in the GET statement.

The GET statement can also be used with the keyboard. An OPEN statement must be executed before the GET statement is encountered. The first argument of the OPEN statement is the number of a channel that is not already OPEN. The channel number must be a number from 1 to 7. The second argument of the OPEN statement must be 4 (input). The third argument is generally 0. The device code "K" is the fourth argument.

With the keyboard, a GET statement causes the program to wait for one keystroke. When a key (or combination of keys) is pressed, the ASCII code of the character is assigned to the variable in the GET statement.

EXAMPLE

```
10 OPEN #2, 4, 0, "K"  
20 GET #2, X  
30 PRINT X  
40 CLOSE #2  
RUN  
(PRESS "S")  
83
```

The previous example consists of a program that uses the GET statement with the keyboard. Line 10 opens channel #2 for the keyboard input. In line 20, the GET statement assigns the ASCII code of a character to the variable X. Line 30 displays the ASCII code on the screen. When the program is executed, line 10 opens the I/O channel, but the program waits at line 20. When the next keystroke occurs, the program continues. In this example, the keystroke is the S key. The ASCII code of S is 83, so X is assigned the value 83. Line 30 causes 83 to be printed on the display, and line 40 closes the I/O channel.

The GET statement can also be used with the display. An OPEN statement must precede the GET statement. The OPEN statement specifies an I/O channel that is not currently open. The channel number must be from 1 to 7. The second argument must be 4 (input) or 12 (input and output), and the device must be "S". With the display, the position of the cursor determines the character or picture element to which the GET statement applies. The GET statement retrieves the COLOR information at that point.

In graphics modes 0, 1, and 2, the COLOR information indicates a character (and color register). Tables 9-4 and 9-7 list the COLOR values for graphics modes 0, 1, and 2. In graphics modes 3 through 8, the GET statement indicates the color of the picture element where the cursor is located. The value that a GET statement retrieves is assigned to the variable in the GET statement. The cursor advances to the next position after a GET statement has been executed. An attempt to execute a GET statement when the cursor is at the last column of the last row results in an error.

EXAMPLE

```

10 OPEN #3, 4, 0, "S"
20 GRAPHICS 2
30 COLOR 65
40 PLOT 0,0
50 POSITION 0,0
60 GET #3, X
70 PRINT X
80 CLOSE #3

```

The previous example consists of a program that uses GET with the display. Line 10 opens I/O channel #3 for input from the display (device "S"). Line 20 specifies graphics mode 2. Line 30 indicates the character and color that is displayed. Table 9-4 lists the COLOR codes for graphics mode 2. COLOR 65 indicates an upper case A in color register 0. Since SETCOLOR is not used in this program, the character is orange, the default color. The PLOT statement at line 40 places the character at the upper left corner of the display. Line 50 moves the cursor to the same position as the character (0,0). The GET statement at line 60 assigns the COLOR information to the variable X. The channel number in the GET statement must be the same as the channel number in the OPEN statement. Line 70 displays the COLOR information (65) on the display, and line 80 closes the I/O channel.

GET can also be used with the screen editor (device "E"). The OPEN statement must include an unused I/O channel number. Also, the OPEN statement must have operation code 4 (input) or 12 (input and output). Since the screen editor uses the keyboard for input, the GET statement has nearly the same function with devices "K" and "E". The GET statement assigns the ASCII code of a keystroke to the variable specified in the statement. The program waits for input from the keyboard before it continues. However, when a GET statement is executed, the character from the keyboard must be followed by RETURN.

EXAMPLE

```
10 OPEN #3, 4, 0, "E"
20 GET #3, X
30 PRINT X
40 CLOSE #3
RUN
(Press "S" followed by RETURN)
83
```

In the previous example, line 10 opens channel #3 for input from the screen editor. When the screen editor is accessed, the screen is cleared. The program will wait at line 20 for input from the keyboard. If more than one character is entered, an error results.

The GET statement only accepts one character, followed by RETURN. If only one character is entered, the GET statement assigns the ASCII code of that character to the variable X. Line 30 displays the value of X which is 83, since the ASCII code of S is 83. Line 40 closes the I/O channel.

GOSUB (GOS.)

GOSUB branches program control to the subroutine beginning at the line number specified by its argument.

CONFIGURATION

GOSUB LN

Subroutines can be called from any part of a program. A RETURN statement, at the end of a subroutine, causes the program to resume execution with the statement directly after the GOSUB statement.

Subroutines are convenient to use when the same set of operations need to be repeated at different parts of a program.

EXAMPLE

```
10 FOR J = 0 TO 2
20 GOSUB 100
30 NEXT J
40 J = 5
50 GOSUB 100
60 END
100 PRINT J;
110 RETURN
RUN
0125
```

The previous example illustrates a subroutine that is called 4 times, from 2 different parts of the program. In this example, only one statement is included in the subroutine. However, many statements can be included in a subroutine.

Line 10 begins a FOR/NEXT loop. The counter (J) is set equal to 0 the first time through the loop. Line 20 calls the subroutine at line 100. As a result, line 100 is executed next. The subroutine prints the value of J and proceeds to line 110. At line 110, the program is returned to the point where the subroutine was called (line 20).

The statement at line 30 is then executed. The NEXT statement causes the loop to be incremented and repeated. The counter (J) is set equal to 1, and the subroutine is called again from line 20. At line 100, the value of J is printed. Line 110 returns the program to line 20.

These steps are also repeated for J = 2. When the loop has been executed 3 times, the program will proceed to line 40. J is assigned the value 5, and the subroutine is called again at line 50. The subroutine prints the value of J. The program then returns to line 60 where it ends.

GOSUB can also be used with ON to branch a program to one of several subroutines.

CONFIGURATION

ON EX GOSUB LN [,LN] [,LN]...

The expression after the ON statement indicates which line number the program proceeds to. This is called the control expression. The control is evaluated and rounded off. If the value is negative or greater than 255, an error occurs. If the value of the control is 1, the program continues at the first line number after GOSUB. If the control is equal to 2, the program continues at the second line number after GOSUB, etc.

If the value of the control is 0 or greater than the number of line numbers, the line after the ON/GOSUB statement is executed.

EXAMPLE

ON X GOSUB 100, 200, 300, 400

This statement executes the subroutine at line 100 if X = 1. If X = 2,

the subroutine at line 200 is executed. If $X = 3$, the subroutine at line 300 is executed. If $X = 4$, the subroutine at line 400 is executed. If $X = 0$ or X is greater than 4, the next line is executed.

GOTO

The GOTO statement causes the program to proceed at the indicated line number.

CONFIGURATION

GOTO LN

EXAMPLE

```
10 X = X + 1
20 IF X ^ 2 > 50 THEN END
30 PRINT X;
40 GOTO 10
RUN
1234567
```

The previous example demonstrates the use of GOTO. Line 10 increases the value of X by 1. Line 20 ends the program when X squared is greater than 50. When line 40 is executed, the program returns to line 10. This program repeats lines 10 through 40 until the program is ended or branched out of the loop. The program ends when $X = 8$ because 8 squared is greater than 50.

GOTO is also used with an ON statement to branch a program to one of several lines.

CONFIGURATION

ON EX GOTO LN [,LN] [,LN]...

The expression after the ON statement indicates which line number the program proceeds to. This is called the control expression. The control is evaluated and rounded off. If the value is negative or greater than 255, an error occurs. If the value of the control is 1, the program continues at the first line number

after GOTO. If the value is 2, the program continues at the second line number after GOTO, etc.

EXAMPLE

```
10 FOR I = 1 TO 3
20 ON I GOTO 40, 50, 60
40 PRINT "I = 1":GOTO 70
50 PRINT "I = 2":GOTO 70
60 PRINT "I = 3"
70 NEXT I
```

GRAPHICS

GRAPHICS sets one of the graphics modes.

CONFIGURATION

GRAPHIC a

The GRAPHICS statement generally clears the screen display upon execution. By adding 32 to the GRAPHICS statement argument, this feature is suppressed.

In graphics modes 1 through 8, a four line text window appears in the bottom of the display. By adding 16 to the GRAPHICS statement argument, the text window will be suppressed.

EXAMPLE

GRAPHICS 49

The preceding GRAPHICS statement sets graphics mode 1 with the screen clearing and text window features suppressed.

IF

The IF statement is used with a THEN statement to branch a program if a particular condition is true.

CONFIGURATION

IF EX THEN ^{statement} LN [:statement]...

The expression (EX) that follows IF can be logical or algebraic. Any algebraic expression that does not equal zero is considered true. The logical operators (AND, NOT and OR) can be used in the IF expression.

EXAMPLE

```
10 X = 15
20 Y = 30
30 IF X>10 AND Y>20 THEN 50
40 PRINT "CONDITIONS NOT MET":END
50 PRINT "CONDITIONS HAVE BEEN MET"
RUN
CONDITIONS HAVE BEEN MET
```

The previous example shows two logical expressions and a logical operator in the IF/THEN statement (line 30). The AND will only be true when both conditions have been met. Since X = 15 (line 10) and Y = 30 (line 20), both of the conditions of line 30 are true. As a result, the program branches to line 50. At line 50, the message CONDITIONS HAVE BEEN MET is printed.

An END statement is used in line 40 to prevent both messages from being printed when the IF statement is false.

An IF/THEN statement can also be followed by statements instead of a line number.

EXAMPLE

```
10 Y = 5
20 X = 10
30 IF X<100 THEN PRINT X:PRINT Y
RUN
10
5
```

The previous example shows that statements can follow a THEN statement, separated by colons. If the condition is true, the statements are executed. If the condition is false, the program will continue at the next line, and the statements after the THEN statement are ignored. Since X = 10 (line 20), the condition at line

30 ($X < 100$) is true. As a result, the statements after THEN are executed, and the values of X and Y are printed.

The following example illustrates the use of algebraic expressions. An algebraic expression is true when it does not equal zero.

EXAMPLE

```
10 FOR I = -2 TO 2
20 IF NOT I THEN END
30 PRINT I
40 NEXT I
RUN
-2
-1
```

The previous example contains a program that ends when a condition is true. The condition is NOT I. NOT I is true when I is false, and I is false when I is set equal to zero. When I has any value other than zero, it is true.

Line 10 begins a FOR/NEXT loop. The first time the loop is executed, I is set equal to -2. Line 20 is an IF/THEN statement with the condition NOT I. When I is set equal to -2, it is considered true because it is not equal to zero. Since I is true, NOT I is false.

The condition at line 20 is false, so the program does not end. Line 30 is executed next, so the value of I is printed. Line 40 returns the program to line 10, where the counter (I) is incremented by 1. I is set equal to -1, so I is still true. Since I is true, NOT I is false. The condition of line 20 fails, so the value of I is printed.

When the loop is executed the third time, I is set equal to zero. I is false, so NOT I is true. Since NOT I is true, the program is ended at line 20.

INPUT (I.)

The INPUT statement causes data to be assigned to variables.

CONFIGURATION

$$\text{INPUT } [\#a,] \begin{matrix} X \\ X\$ \end{matrix} \begin{bmatrix} Y \\ Y\$ \end{bmatrix} \dots$$

The INPUT statement is generally used with the keyboard, editor, disk, or Program Recorder. The INPUT statement requires an I/O channel number as well as a previous OPEN statement if any device other than the editor is used.

The correct format for numeric data is standard notation or scientific notation. Spaces can appear before or after a numeric value, but spaces within a numeric value cause an error. Numeric data can be entered on the same line, separated by commas.

EXAMPLES

```
54, 4E5, -10
-3.45E-10
0,1,1,5,3,10
```

Expressions cannot be used as numeric data with INPUT. Any format other than standard floating point decimal or scientific notation causes an error.

Each line of numeric data must be followed by an end-of-line character (RETURN).

String data must also be followed by an end-of-line character. Only one string data item can occur on a line. Also, a string data can be read only into dimensioned string variables. If the length of a data item is more than the dimensioned length of the variable, the excess characters are eliminated, but no error occurs. Any character can be a part of a string data item for INPUT (including commas and special graphics characters).

When INPUT is used with the screen editor, no OPEN statement is necessary. The program waits for input from the keyboard when an INPUT statement is executed. A question mark (?) appears on the screen to remind the operator to enter data.

EXAMPLE

```
10 DIM X$(10)
20 INPUT X, X$
30 PRINT X$, X
RUN
? 45, JONES, BILL
JONES, BILL          45
```

In the previous example, line 10 dimensions the string variable for 10 characters. Line 20 is an INPUT statement that requests a numeric value to assign to X, and a string value to assign to X\$. When the program is executed, the INPUT statement causes the program to wait at line 20 for input.

Since no I/O channel is specified, the input is accepted from the keyboard, and the prompt (?) is displayed. The user responds with two data items. The value 45 is entered for a value of X. The string value JONES, BILL is entered for a value of X\$. These two data items could be entered on separate lines. Notice that the comma in the string value does not separate data items.

When each variable in the INPUT statement is assigned a value, the program executes the NEXT statement (line 30). At line 30 the values of X\$ and X are displayed on the screen.

The INPUT statement can also be used with the Program Recorder to recover data. When the Program Recorder is used, an OPEN statement must be executed before an INPUT statement is encountered. The OPEN statement must include an I/O channel number, the operation code for input (4), and the device code ("C"). The third argument of the OPEN statement is a special function code, and must be zero. If any of the arguments of an OPEN statement are not integers, they are rounded off.

The INPUT statement recovers data that was stored with the PRINT statement.

EXAMPLE

```
10 DIM A$(100)
20 OPEN #1, 4, 0, "C"
30 INPUT #1, A$
40 PRINT A$
50 CLOSE #1
```

The previous example contains a program that reads and displays one string value. Line 10 dimensions the variable A\$. Line 20 opens I/O channel #1 for input from the Program Recorder. When line 20 is executed, the tone sounds to remind the operator to find the correct position on the tape, press PLAY on the Program Recorder then press RETURN on the keyboard.

When line 30 is executed, one string value is read from the cassette and assigned to the variable A\$. Line 40 causes the value of A\$ to be displayed on the screen. Line 50 closes the I/O channel.

Before an INPUT statement can be used with the Program Recorder, the data must have been put on the cassette with a PRINT statement.

The INPUT statement can also be used to recover data that was saved on a disk. The INPUT statement has the same configuration with the disk and cassette. The INPUT statement must include an I/O channel number and variable names.

The OPEN statement for the I/O channel must include the channel number and the operation code 4 (input) or 12 (input and output). The third argument of the OPEN statement is zero, and fourth argument is the device and filename.

EXAMPLES

```
OPEN #2, 4, 0, "D2:BUDGET.BAS"  
OPEN #3, 12, 0, "D:NAMES"
```

If only one disk is in use, the device name is simple "D:". If 2 or more disks are being used, the number of the disk must be specified.

The INPUT statement can also be used with the keyboard. The OPEN statement must include an I/O channel number, operation code 4, special operation code 0, and the device "K".

EXAMPLE

```
10 DIM Y$(10)  
20 OPEN #2, 4, 0, "K"  
30 INPUT #2, X, Y$  
40 PRINT X, Y$  
50 CLOSE #2
```

The previous example contains a program that uses the keyboard for input. Line 10 dimensions the variable Y\$. Line 20 opens I/O channel #2 for input from the keyboard. When line 30 is executed, the program waits for input. However, no prompt symbol appears, and the data is not displayed when it is entered.

The first variable in the INPUT statement is X. Since X is a numeric variable, a numeric data item must be entered first. The second variable in the INPUT statement is Y\$. Since Y\$ is a string variable, a string data item must be entered next. A comma can be used to separate the data items, or each data item can be followed by RETURN.

Line 40 displays the values of the two variables, and line 50 closes the I/O channel.

INT

The INT function returns the largest integer that is less than or equal to the argument.

CONFIGURATION

X = INT (a)

EXAMPLES

PRINT INT (13.9)

13

PRINT INT (-4.7)

-5

LEN

The LEN function returns the number of characters in a string value or variable, including spaces and punctuation.

CONFIGURATION

X = LEN (string)

EXAMPLE

```
10 DIM A$(20)
20 A$ = "JONES, BILL"
30 PRINT LEN(A$)
40 PRINT LEN("BILL JONES")
RUN
10
10
```

Line 10 dimensions the variable A\$, and line 20 assigns A\$ a string value. Line 30 displays the number of characters in the variable A\$. Line 40 displays the number of characters in the string "BILL JONES".

LET (LE.)

The LET statement is optional. It is used to assign a value to a variable.

CONFIGURATION

$$[\text{LET}] \quad \begin{matrix} X \\ X\$ \end{matrix} = \begin{matrix} a \\ a\$ \end{matrix}$$
EXAMPLES

```
LET X = 250
X = Y + 25
```

LIST (L.)

The LIST statement is used to display or record information in the computer's memory.

CONFIGURATION

LIST [device:filespec,][LN][,LN]

The LIST statement can be used to save a program, or part of a program, on a disk or cassette. The ENTER statement is the only Atari BASIC statement that can recover a program saved with LIST. The optional line numbers (LN) indicate the section of the program that is to be saved. If no line numbers are specified, the entire program will be saved. If only one line number is specified, only that line of the program is saved. If two line numbers are specified, those two lines are saved along with all the code between those line numbers. If either or both of the specified line numbers do not appear, the section of the program between those line numbers is saved.

A program is saved on a cassette tape with the statement LIST "C". Before saving the program, the tape must be properly positioned. When a LIST "C" statement is executed, the tone sounds twice to remind the operator to press PLAY and RECORD on the Program Recorder, followed by RETURN on the keyboard.

DOS must be booted before a LIST statement can be used with a disk. A program is saved on a disk with a statement of the form LIST "device:filespec" followed by the appropriate line numbers (if any).

EXAMPLE

```
10 DIM A$(10)
20 FOR A = 1 TO 100
30 PRINT A$, A ^ 2
40 IF A ^ 2 > 500 THEN END
50 NEXT A
LIST "D:PROGR.BAS", 5, 45
```

In the previous example, the LIST statement saves lines 10 through 40 on the disk. The line numbers that are specified (5 and 45) do not exist in the program, so the section of the program with line numbers between those values is saved.

The device code "D:" can be used only if one disk is in use. If more than one disk is available, the number of the disk must also be specified.

The LIST statement can also be used to display a program on the monitor. The LIST command displays the entire program on the screen unless the LIST statement is followed by line numbers.

If one line number follows the LIST statement, the line of the program with that line number is displayed. If the program does not have a line with the line number specified in the LIST statement, the LIST statement has no results.

EXAMPLE

```
LIST 20

20 FOR A = 1 TO 100

READY
```

If two line numbers are specified, those two lines are displayed along with all the code between those line numbers. If either or both of the specified line numbers do not appear in the program, the section of the program between those line numbers is displayed.

The LIST statement can also be used with a printer. The statement

LIST "P:" causes the program in the computer's memory to be listed on the printer. The interface module and the printer must both be turned on. Also, the printer must be online.

The computer's character set is slightly different from the printer's, so certain characters appear differently when printed. Also, the printer interprets some of the control characters as commands. As a result, when control characters are printed, the printer may have an unusual response. To avoid this problem, do not use control characters within quotation marks. Instead, use the CHR\$ function to generate special characters.

EXAMPLE

```
PRINT "→" (escape, control - *)
PRINT CHR$(31) (preferred)
```

The computer can only accommodate 128 variables. If the limit is exceeded, ERROR-4 occurs. The computer maintains a variable name table with the names of all variables used since the NEW command was executed. As a result, the variable name table can accumulate variable names that are no longer being used. The LIST statement is the only Atari BASIC statement that saves a program without saving the variable name table. As a result, the LIST and ENTER statements can be used to eliminate unused variables from the variable name table.

EXAMPLE

Save the program on cassette or disk using LIST.
Execute a NEW statement to clear the memory.
Put the program back into memory using ENTER.

LOAD (LO.)

The LOAD statement is used to recover programs that were recorded with the SAVE statement.

CONFIGURATION

```
LOAD "device:filespec"
```

The LOAD statement is used with the Program Recorder or a disk. The LOAD statement can only be used to recover programs that were previously saved with a SAVE statement.

When a LOAD statement is executed, the computer's memory is cleared before the new program is loaded. Also, the I/O channels are closed (except 0), and the sound voices are shut off.

With the Program Recorder, the LOAD statement does not use a filename. The cassette tape must be correctly positioned before the LOAD statement is executed. Only the device name "C" is necessary. When the LOAD "C" statement is executed, the tone sounds once to remind the operator to press PLAY on the Program Recorder, followed by RETURN on the keyboard.

With a disk, the LOAD statement must include a device name along with a filename. If more than one disk is in use, the device name must also include the number of the disk. If only one disk is in use, the device name "D:" is sufficient.

EXAMPLE

LOAD "D2:GRADES"

LOCATE (LOC.)

The LOCATE statement is used to place the cursor at the specified position, and assign the COLOR data at that point to the specified numeric variable.

CONFIGURATION

LOCATE a, b, X

The first argument (a) indicates the column that the cursor is moved to. The second argument (b) indicates the row. The third argument is the numeric variable that is assigned the COLOR data at the cursor position. A LOCATE statement can only be used if a GRAPHICS statement has been executed.

The COLOR data in graphics mode 0 corresponds to the

character graphics mode 1 and 2, the COLOR data indicates the character and color register of a PLOT statement.

In graphics modes 3 through 8, the COLOR data actually corresponds to the color register of a picture element.

EXAMPLE

```
10 GRAPHICS 3
20 COLOR 2
30 PLOT 0,0
40 DRAWTO 35,0
50 LOCATE 5, 0, X
60 PRINT X
```

The previous example consists of a program that uses the LOCATE statement. Line 10 chooses graphics mode 3. Line 20 indicates that color register 2 is used in the PLOT and DRAWTO statements. Since no SETCOLOR statement was executed, the default color (green) is used. The PLOT statement at line 30 illuminates a green picture element at the upper left corner of the screen. The DRAWTO statement at line 40 illuminates the top row of the display in the same color. Line 50 is a LOCATE statement that places the cursor at position 5,0. Since the line was drawn from 0,0 to 35,0, the position 5,0 is an illuminated picture element. The value of the color register at that position is 2. The LOCATE statement assigns the color register value (2) to the variable X. Line 60 is a PRINT statement that displays the value of X.

The DRAWTO and XIO statements have separate memory locations for the cursor position. As a result, a LOCATE statement has no effect on the cursor position of a DRAWTO or XIO statement.

When LOCATE is used to read a code from the screen, the cursor will move one location to the right. If the cursor was on that last column of a row when LOCATE was executed, the cursor may attempt to advance to the first column of the next row resulting in Error 141 (Cursor Out of Range).

LOCATE moves the cursor by altering the values stored in memory address 84 (current cursor row number) and memory addresses 85 and 86 (current cursor column number). The cursor position change as a result of the execution of LOCATE will have no effect on DRAWTO and XIO statements, as they use memory addresses 90, 91, and 92 to determine the next cursor address.

LOG

The LOG function returns the natural logarithm of the argument. The natural log function is undefined for arguments less than or equal to zero.

CONFIGURATION

$X = \text{LOG}(a)$

EXAMPLES

```
PRINT LOG(2.71828183)
1
PRINT LOG (-1)
ERROR-3
```

A value error results from a zero or negative argument.

LPRINT (LP.)

The LPRINT statement sends a line of output to a printer.

CONFIGURATION

LPRINT [data] ; [data]...

The LPRINT statement can include numeric variable names and string variable names, as well as string constants. String constants must appear in quotation marks.

The items in an LPRINT statement must be separated by a comma

or a semicolon. A semicolon causes the values to be printed on the same line without any spaces. A comma causes the next item to be printed at the next column stop location. A comma or semicolon is optional at the end of a LPRINT statement. If a semicolon is used at the end of a LPRINT statement, the next output will be adjacent to the last output. If a comma is used at the end of an LPRINT statement, the next output occurs at the next column stop after the last output. If neither a comma nor a semicolon is used at the end of an LPRINT statement, the next output occurs on the next line.

When an LPRINT statement is executed, an error occurs if the printer is not ready to operate.

The LPRINT statement uses I/O channel 7. If channel 7 is open when an LPRINT statement is executed, an error will occur.

EXAMPLE

```
10 DIM A$(5)
20 A$ = "GREEN"
30 X = 25
40 LPRINT "INVENTORY: ";X,A$
```

In the previous example, LPRINT is used to print a string constant, a string variable, and a numeric variable. The LPRINT statement at line 40 prints the word INVENTORY followed by a colon and a space. Any characters that appear in quotation marks are reproduced as they appear. A semicolon separates the items, so the value of X (25) follows the string.

A comma separates the variable names X and A\$, so the value of A\$ is printed in the next display column.

NEW

The NEW command eliminates the current program in the computer's memory. The NEW command erases all variables, turns off all voices, and closes all I/O channels except channel 0.

CONFIGURATION

NEW

EXAMPLE

NEW

NEXT (N.)

The NEXT statement is used with a FOR statement to form a repetitive section of a program.

CONFIGURATION

NEXT X

A FOR statement begins a loop, and a NEXT statement ends it. The FOR statement sets an initial value and a final value for the counter. The optional STEP statement specifies the amount that the counter is increased or decreased each time the loop is executed.

EXAMPLE

```
10 FOR I = 1 TO 10 STEP 2
20 PRINT I
30 NEXT I
```

In the previous example, the variable I is the counter. The initial value of the counter is 1, and the final value is 10. The value of the counter is incremented by 2 each time the loop is executed.

The section of the program between the FOR and NEXT statements is repeated for each different value of the counter. Each time the NEXT statement is executed, the value of the counter is changed by the STEP argument value. The loop is repeated for each value of the counter. In the previous example, the loop is repeated 5 times, with the counter equal to 1, 3, 5, 7, and 9. The initial value of the counter (I) is 1, and it is increased by 2 each time the loop is executed because of the STEP 2 statement.

If no STEP statement is used, the counter value increases by 1 each time a NEXT statement is executed.

A FOR/NEXT loop can also have a decreasing counter. If the STEP argument is negative, the value of the counter decreases each time the loop is executed.

An increasing counter will repeat the loop until one more increase would make the counter greater than the final value. A decreasing counter will repeat the loop until one more decrease would make the counter less than the final value.

When a loop has been completed, the statement after the NEXT statement is executed.

NOT

NOT is a logical operator that returns the value 1 if its argument is false. If its argument is true, the NOT statement returns the value 0.

CONFIGURATION

$X = \text{NOT } EX$

The following truth table describes the NOT operator.

A	NOT A
0	1
1	0

The computer represents the condition of true with the number 1. The false condition is represented by 0.

Numbers and expressions are considered true if they equal any number other than 0. Only numbers that equal 0 are false. The following examples are true.

EXAMPLES

```

5>3
4
NOT 0
NOT 3>5

```

The following examples are false.

EXAMPLES

```

"DOG" = "CAT"
3>5
NOT 5
NOT 1

```

The NOT operator is generally used in IF/THEN statements.

EXAMPLE

```

If X>Y AND NOT Z THEN 250

```

NOTE (NO.)

The NOTE function returns the location of the file pointer for a specified disk file. The NOTE function is not available in DOS version 1.0.

CONFIGURATION

```

NOTE #a, X, Y

```

The NOTE function must specify a channel number (#a) that is open for a disk file.

The second argument is a numeric variable that is assigned the sector number of the file pointer. The third argument is a numeric variable that is assigned the byte number of the file pointer within the specified sector.

EXAMPLE

```

NOTE #2, SEC, BYT

```

ON

The ON statement is used to branch program control. When used with a GOTO statement, the ON statement branches program control to one of several lines. An ON statement is also used with GOSUB to branch a program to one of several subroutines.

CONFIGURATION

$$\text{ON X } \begin{bmatrix} \text{GOSUB} \\ \text{GOTO} \end{bmatrix} \text{LN[,LN]...}$$

The argument of ON is the control expression. When a GOSUB statement is used, the program proceeds to a subroutine. When a GOTO statement is used, the program branches to a line number.

The control expression determines to which line number the program will proceed. If the control expression equals 1, the program branches to the first line number after the GOTO or GOSUB. If the control expression equals 2, the program branches to the second line number after GOTO or GOSUB, etc.

If the control expression does not equal an integer, it is rounded. If the control expression evaluates to 0 or a number greater than the number of choices of line numbers, the statement following the ON statement is executed.

If the control expression is less than 0 or greater than 255, an error results.

EXAMPLE

```
10 X = 2
20 ON X GOTO 30, 40, 50
30 PRINT "FIRST":END
40 PRINT "SECOND":END
50 PRINT "THIRD":END
RUN
SECOND
```

The previous example consists of a program that uses an ON/GOTO branch. At line 20, the ON/GOTO statement branches to line 30, 40, or 50 depending on the value of X. Since X is assigned the value 2, the ON/GOTO statement causes a branch to the second line number. The second choice is line 40, so the message SECOND is printed.

OPEN (O.)

The OPEN statement is used to open an input/output channel for an input or output device. The computer cannot receive input from or send output to a device unless an I/O channel has been opened for that purpose.

CONFIGURATION

OPEN #a, b, c, "device [:filespec]"

The first argument of an OPEN statement is the channel number. The channels are numbered from 0 through 7. Channel number 0 is always reserved for the editor. Channel number 6 is used for graphics, and channel number 7 is used to save and load programs. Channel number 7 is also used with the LPRINT statement.

As a result, channels 1 through 5 are available for use with BASIC programs. Channels 6 and 7 are available only on a limited basis for use with BASIC programs. Channel 6 is available if no graphics are used. Channel 7 is available unless programs are being loaded or saved. Also, channel 7 is unavailable if an LPRINT statement is executed.

The second argument indicates the operation of the input/output device. In general, the second argument is 4 if the computer is accepting information (input). The second argument is generally 8 if the computer is sending information (output) to a device. Table 5-1 contains a complete list of I/O operations with their associated devices and operation numbers.

Table 5-1. I/O Operations

Device	Operation Number	Operation Type
Program Recorder	4	input
	8	output
Keyboard	4	input
Printer	8	output
Editor	8	output:screen
		input:keyboard
	12	output:screen
		input:screen
	13	output:screen
Disk	4	input
	6	read disk directory
	8	output, new file
	9	output, append
	12	input and output, update
Interface	5	concurrent input
	8	block output
	9	concurrent output
	13	concurrent input and output

The third argument of an OPEN statement indicates a special operation. The special operation code is usually 0. Generally, the third argument is only used when opening the screen display for a graphics mode.

If any of the first three arguments of an OPEN statement are not integers, they are rounded off.

The fourth argument of an OPEN statement is the device name. The device names used by Atari computers are listed below. The device name and file specification (if present) must be enclosed in quotation marks.

Program Recorder	C:
Screen Editor	E:
Keyboard	K:
Printer	P:
Display	S:
Disk	D:

Program Recorder

An I/O channel can be opened for the Program Recorder for either input or output, but not both at the same time. When the OPEN statement is executed, the tape must be at the correct location before proceeding.

When an OPEN statement is executed for output to the Program Recorder, the tone sounds twice. This is a reminder for the operator to press Play and Record on the Program Recorder, followed by Return on the keyboard. For input, the tone sounds once to remind the operator to press Play on the Program Recorder, followed by Return on the keyboard.

The third argument of an OPEN statement for the Program Recorder can be either 0 or 128. The files are recorded with shorter gaps between the records when the third argument is 128.

When an OPEN statement is executed, and the correct levers on the Program Recorder are pressed, the Program Recorder begins operating as soon as the Return key on the keyboard is pressed. The tape keeps moving until a set of data (128 bytes) is accumulated for output. While data is being accumulated, nothing is recorded on the tape. As a result, if a long delay occurs from the period when the OPEN statement is executed to when the information is recorded, a long gap appears on the tape.

When a long section of blank tape (30 sec. or more) is encountered during input, a Device Timeout error occurs. To avoid these errors, the I/O channel should be closed whenever a delay in the output procedure occurs.

Keyboard

The OPEN statement for the keyboard can be for input only. When the keyboard is used for input, the question mark does not appear as a prompt for an INPUT statement. Also, the response to an INPUT statement does not appear on the display. The third argument of an OPEN statement for the keyboard is ignored.

EXAMPLE

```
10 DIM A$(1)
20 OPEN #2, 4, 0, "K:"
30 GRAPHICS 3 + 16
40 INPUT #2, A$
50 PRINT
60 END
```

The previous example contains a program that maintains a graphics display until input is received from the keyboard. Line 10 dimensions the string variable A\$. Line 20 opens the keyboard for input. Line 30 selects graphics mode 19, which is the same as graphics mode 3, but without a text window.

In order to maintain a full screen graphics display, the program must pause, but not end. When a character is displayed, the display returns to graphics mode 0.

When the INPUT statement is executed at line 40, the program waits for input, but does not ruin the display by printing the prompt (?) or the response. As a result, the display is preserved until the operator enters a suitable input for A\$. The easiest response to the INPUT statement is the Return key.

Disk

An I/O channel can be opened for a disk for any of the I/O operations listed in Table 5-1. When an OPEN statement for the disk is executed, DOS must have been booted and ready to operate.

An OPEN statement for a disk file must include the filename and

optional filename extension. The filename extension must be separated from the filename by a period.

The following examples are correct OPEN statements for a disk.

EXAMPLE

```
OPEN #1, 4, 0, "D2:GRADES.BAS"  
OPEN #3, 12, 0, "D:JONES"
```

Printer

An I/O channel for the printer can be for output only. The printer must be turned on before the OPEN statement is executed. If the printer is used with the Atari 850 interface, this also must be ready to operate. The printer must be in the Online mode if it has Local/Online switch.

The third argument of an OPEN statement for the printer is generally 0. However, the Atari 820 printer outputs sideways characters if the third argument is 83.

Editor

An OPEN statement for the editor allows the screen and keyboard to be used for input and output. When an OPEN statement is executed for the editor, the display resumes graphics mode 0, the screen is cleared, the cursor is reset, and the color registers are set to the default values.

The editor can be used in one of three modes. The mode is determined by the second argument of the OPEN statement (Table 5-1). The display is always used for output, but the display or the keyboard can be used for input.

The third argument of an OPEN statement for the editor is ignored. Even though this value has no effect, it must always be included in the OPEN statement.

EXAMPLE

```
10 OPEN #1, 13, 0, "E:"  
20 T = 3.14  
30 PRINT T  
40 POSITION 0,0  
50 INPUT #1, X  
60 PRINT X  
70 END
```

The previous example contains a program that uses the display screen as an input device. Line 10 opens I/O channel number one for the editor (device "E:"). The second argument of the OPEN statement (13) indicates that the display is used for input and output. The second line of the program assigns the value 3.14 to the variable T. Line 30 causes the value of T to be displayed on the screen. Since the OPEN statement clears the screen and resets the cursor, the value 3.14 is displayed at the upper left hand corner of the screen.

The POSITION statement at line 40 returns the cursor to the upper left hand corner of the screen. The INPUT statement at line 50 chooses the device on I/O channel 1. As a result, the screen is used to input a value for the variable X.

When an INPUT statement is used with the screen, the value that follows the cursor is used for input. Since the value 3.14 appears at the top of the screen, and the cursor is also at the top of the screen, the value 3.14 is assigned to X. Line 60 displays the value of the variable X.

The output of this program is the value 3.14 displayed twice. The number is repeated because it is printed at lines 30 and 60.

Screen

The OPEN statement for the screen (device "S:") is used to choose a graphics mode. The third argument of the OPEN statement indicates the graphics mode (0 through 8). The second argument indicates if the screen is used for input or output, or both. Also, the second argument determines if the display has a text window and if the display is cleared when the OPEN statement is executed.

Table 5-2. Screen I/O Operations

OPERATION NUMBER	OUTPUT	INPUT	TEXT WINDOW	CLEAR SCREEN
8				
12				
24				
28				
40				
44				
56				
60				

NOTE: The screen is always clear in graphics mode 0.
Graphics mode 0 has no separate text window.

When the screen is used with an OPEN statement instead of a GRAPHICS statement, the PLOT and DRAWTO statements cannot be used. Input and output is performed with PRINT, PUT, and GET statements. Each of these statements require an I/O channel number that corresponds to the OPEN statement channel number.

EXAMPLE

```

10 GRAPHICS 8
20 COLOR 1
30 PLOT 0,0
40 DRAWTO 10,10
50 OPEN #1, 60, 8, "S:"
60 POSITION 5,5
70 GET #1, X
80 PRINT X
90 END

```

The previous example contains a program that uses the screen as an input device. Line 10 has a GRAPHICS statement that indicates graphics mode 8. Line 20 chooses color number 1. Lines 30 and 40 draw a small diagonal line in the upper left of the display.

At line 50, the display is opened as an I/O device. The first argument of the OPEN statement indicates the I/O channel number. The second argument indicates that the screen is used for input and output. Also, a text window is present, and the screen is not cleared (Table 5-2). The third argument of the OPEN statement indicates graphics mode 8.

At line 60, the cursor is positioned at the location of 5,5. The GET statement at line 70 assigns the color number at the cursor position to the variable X. Since the cursor is at location 5,5, the color number at that location is 1 (5,5 is one of the points on the line between 0,0 and 10,10). The PRINT statement at line 80 displays the value of the variable X in the display window.

Atari 850 Interface Module

An OPEN statement for a serial port of an Atari 850 Interface module requires the device name "R:". The number of the port is also necessary for ports 2 through 4. The first argument of the OPEN statement is the I/O channel. The second argument determines the I/O operation, as listed in Table 5-1. The third argument is ignored. Although the third argument has no effect, it must appear in the OPEN statement.

The interface module must be ready to operate when the OPEN statement is executed. It will not operate unless it was turned on before the computer console was turned on. Also, the interface module may not operate properly until the appropriate XIO statements have been executed.

The following examples are correct OPEN statements for the interface module.

EXAMPLES

```
OPEN #1, 5, 0, "R2:"  
OPEN #2, 13, 0, "R:"  
OPEN #4, 8, 0, "R4:"
```

OR

The OR statement is a logical operator that returns the value 1 if either one of its arguments are true. An OR statement returns the value 0 only if both of its arguments are false.

CONFIGURATION

EX OR EX

The conditions of true and false are represented by the values 1 and 0 respectively. The results of the OR operation are represented by the following truth table.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

An OR statement can have either relational or algebraic expressions for arguments. Any algebraic expression that does not equal zero is true. An expression that equals zero is considered false.

EXAMPLES

5 (true)
 3-7 (true)
 "DOG" = "CAT" (false)
 8<2 (false)

In the previous examples, 5 is considered true because it does not equal zero. The expression 3-7 is also true because it does not evaluate to zero. The relational expression "DOG" = "CAT" is false because the string constants are not equal. The expression 8<2 is also false.

EXAMPLE

```
10 X = 5
20 Y = 10
30 IF X = 10 OR Y THEN PRINT Y
40 END
RUN
10
```

The previous example consists of a program that uses an OR statement within an IF/THEN statement. Line 10 sets X equal to 5. Line 20 sets Y equal to 10. Line 30 displays the value of Y if either (or both) of the arguments of the OR statement are true. The first argument of the OR statement is the relational expression $X = 10$. Since X is set equal to 5 in line 10, this expression is false. The second argument of the OR statement is the algebraic expression Y. The expression (Y) is considered false only when it equals zero. Since Y is set equal to 10 at line 20, the expression is considered true.

As a result, the OR statement is true because one of the arguments is true. The value of Y is displayed because the condition of the IF/THEN statement is true.

PADDLE

The PADDLE function returns an integer between 1 and 228 that depends on the rotation of a particular paddle.

CONFIGURATION

$X = \text{PADDLE (a)}$

A total of 8 paddle game controllers can be used at one time. The value of the argument (a) indicates the number of the paddle. If the argument of the PADDLE statement is not an integer, it is rounded off. The paddles are numbered 0 through 7. If the PADDLE statement has an argument greater than 7, the results are unpredictable. If a paddle is not present when a PADDLE statement is executed, the value 228 is returned.

The paddle controllers are used only in pairs. A pair of controllers is plugged into one of the controller jacks on the front of the computer. The first jack accepts paddles 0 and 1. The second jack accepts paddles 2 and 3, etc.

If a paddle is rotated fully clockwise, the value 1 is returned. The value increases as the paddle is rotated counter-clockwise. The maximum value returned is 228.

EXAMPLE

```
10 IF PADDLE (1)=150 THEN END
20 GOTO 10
```

The previous example consists of a program that executes line 10 repeatedly until the paddle is rotated more than halfway counter-clockwise. Since PADDLE (1) specified, the paddles must be plugged into controller jack 1.

PEEK

The PEEK function is used to recover the value in a memory location.

CONFIGURATION

$X = \text{PEEK } (a)$

A memory location contains an integer value between 0 and 255. The argument of a PEEK statement refers to the memory location. A value error occurs if the argument is negative or greater than 65535. If the argument (a) is not an integer, it is rounded off.

Many memory locations are of general interest. The contents of a memory location can be changed with a POKE statement. Appendix E contains information about commonly used memory locations.

EXAMPLE

```
PRINT PEEK (83)
39
```

The previous example displays the current value for the right margin of the screen. The default value is 39.

PLOT (PL.)

The PLOT statement is used to display a character or picture element on the display.

CONFIGURATION

PLOT a,b

The arguments of a PLOT statement determine the position on the screen where the character or picture element appears. The first argument (a) indicates the column, and the second argument (b) indicates the row. The graphics mode specified determines the number of rows and columns of the display. If either of the arguments is not an integer, it is rounded off. If either argument is negative or greater than the dimension of the screen an error results.

In graphics mode 0, the COLOR statement indicates the character that will appear at the next PLOT location. However, the COLOR statement has nothing to do with the color of the character. Table 9-7 indicates the COLOR value for each character.

In graphics modes 1 and 2, the COLOR statement indicates the character and location register used in the next PLOT location. Table 9-4 indicates the COLOR value for each character in each color register. In graphics modes 1 and 2, color registers are available for each character. As a result, each character can be displayed in any of 4 colors.

In graphics modes 3 through 8, the PLOT statement illuminates a picture element at the screen position indicated by the arguments of the PLOT statement. The dimensions of the display depend on the graphics mode. The number of possible colors also depends on the graphics mode.

EXAMPLE

```
10 GRAPHICS 3
20 COLOR 2
30 FOR I = 0 TO 35 STEP 5
40 PLOT I,0
50 NEXT I
```

The previous example contains a program that uses a PLOT statement. Line 10 indicates graphics mode 3. Line 20 chooses the color register 2. Since no SETCOLOR statement was executed, color register 2 defaults to green. Line 30 begins a FOR/NEXT loop that is executed 8 times. The value of the counter (I) is set to equal 0, 5, 10, 15...35. As a result, line 40 causes green picture elements to appear evenly spaced across the top of the display. The PLOT statement indicates positions (0,0), (5,0), (10,0)...(35,0). The first argument is the column, and the second argument is the row. The second value indicates the top line (zero row) of the display.

POINT (P.)

The POINT statement is used only in disk operations to move the file pointer to a given location.

CONFIGURATION

POINT #a, b, c

The first argument of a POINT statement indicates an I/O channel. The channel must be open to a disk for input, update, or append. The second argument is the sector value. The sector value must lie within the limits of the file. The third argument is the number of the byte within the sector. The third argument must be between 0 and 125. If any of the arguments are not integers, they are rounded off.

The POINT command is not valid in version 1.0 of the disk operating system.

EXAMPLE

POINT #3, SECT, BYTE

In the preceding example, the file pointer for the disk file opened through channel #3 is moved to the sector specified by the variable, SECT, and byte within that sector specified by the variable, BYTE.

POKE (POK.)

The POKE statement is used to store one byte of information in a particular memory location.

CONFIGURATION

POKE a, b

The first argument of a POKE statement is the memory location. If a POKE statement specifies a memory location that does not exist, the POKE statement has no effect. Also, if a POKE statement specifies a memory location that is part of the ROM, the POKE statement has no effect.

The second argument of a POKE statement is the value that is to be stored at the specified memory location. The value of the second argument represents one byte. As a result, the value must be an integer between 0 and 255.

If either of the arguments of a POKE statement is not an integer, it is rounded off. A value error occurs if the memory location specified is greater than 65535 or the value of the second argument exceeds 255. A value error also results if either of these arguments are negative.

If the POKE statement is not used carefully, it can seriously disrupt the operation of the computer.

Appendix E contains information regarding commonly used memory locations.

EXAMPLE

POKE 83,20

The previous example consists of a statement that changes the right margin of the screen to column 20. The value of the right margin is stored in memory location 83.

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.

Since the POP statement was executed, the program is no longer in the subroutine. If another RETURN statement is executed, the program will not return to line 30, where the subroutine was called. The program will return to the line of the previous GOSUB statement. Since there is no other GOSUB statement in this program, a RETURN statement would cause an error.

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.

POSITION (POS.)

The POSITION statement moves the cursor to the specified column and row.

CONFIGURATION

POSITION a, b

The first argument of the POSITION statement determines the column, and the second argument determines the row. The cursor does not actually move when the POSITION statement is executed. The cursor takes on the new position when the next PUT, GET, PRINT, INPUT, or LOCATE statement is executed.

If a POSITION statement specifies a location that is outside the range of the display, no error occurs until another statement that

uses the display is executed.

A POSITION statement does not affect the DRAWTO, PLOT, or XIO functions. These operations maintain a separate cursor location.

EXAMPLE

```
10 GRAPHICS 0
20 POSITION 5, 4
30 PRINT EXP(1)
```

The previous example contains a program that uses a POSITION statement. The GRAPHICS 0 statement causes the display to be cleared. Line 20 moves the cursor to column number 5 and row number 4. Line 30 prints the output on the screen at the position of the cursor. As a result, the value 2.71828179 is displayed four lines from the top of the display and 5 columns from the left margin.

PRINT (PR. or ?)

The PRINT statement is generally used to display characters on the screen, but a PRINT statement can be used to output characters to any output device.

CONFIGURATION

PRINT[#a;] [expression][?]...

The PRINT statement can include numeric variable names and string variable names, as well as string and numeric constants. String constants must appear in quotation marks.

Items within a PRINT statement must be separated by a comma or a semicolon. A semicolon causes the values to be printed on the same line, without any spaces between items. A comma causes the next item to be printed at the next column stop location.

If a semicolon is used at the end of a PRINT statement, the next PRINT statement output will be adjacent to the last output. If a comma is used at the end of a PRINT statement, the next output occurs at the next column stop after the last output. If neither a comma nor a semicolon is used at the end of a PRINT statement, the next output occurs on the next line.

Column stops occur at intervals of 10 spaces. However, if the last character that was printed is within two spaces of the next column stop, that column stop will be ignored. As a result, items in a PRINT statement that are separated by commas will have at least two spaces between them.

EXAMPLE

```
10 DIM A$(15)
20 A$ = "THOMAS R SMITH"
30 X = 27
40 PRINT "NAME: "; A$, "AGE: "; X
50 END
```

The previous example contains a program that uses a PRINT statement. At line 10, the variable A\$ is dimensioned. At line 20, the variable A\$ is assigned the string value "THOMAS R SMITH". At line 30, the variable X is assigned the value 27.

Line 40 contains a PRINT statement. The string constant "NAME:" is printed first, followed immediately by the value of the variable A\$. Since a comma follows the variable A\$, the string constant "AGE:" is printed in the next available column. However, the last character was printed in column 19, so the column stop at column 20 is ignored. As a result, the string constant "AGE:" and the value of the variable X are displayed in the last column.

A PRINT statement requires an I/O channel number for any output device other than the display. The I/O channel must be open for an appropriate output operation.

Program Recorder

A PRINT statement that is used with the Program Recorder is generally used to store data that will be recovered with an INPUT statement.

If the OPEN statement for the Program Recorder specifies short gaps between the records (special operation code 128), the tape does not stop moving. The data is not recorded correctly if the program does not supply data fast enough to keep up with the tape.

EXAMPLE

```
10 OPEN #1, 8, 0, "C:"  
20 FOR I = 1 TO 100  
30 X = INT (RND (9) * 100)  
40 PRINT #1, X  
50 NEXT I  
60 CLOSE #1  
70 END
```

The previous example contains a program that records 100 random numbers on tape. Line 10 opens I/O channel number 1 for output to the Program Recorder. At line 20, a FOR/NEXT loop is set up to be repeated 100 times. Line 30 assigns a random number between 0 and 99 to the variable X. At line 40, the value of X is printed on tape, using the Program Recorder. Line 50 is the NEXT statement that completes the FOR/NEXT loop. Line 60 closes the I/O channel, and line 70 ends the program.

Disk

A PRINT statement that is used with a disk is generally used to store data that will be recovered with an INPUT statement.

The format for using a PRINT statement with a disk is the same as with the Program Recorder. The appropriate OPEN statement must precede the PRINT statement. The I/O channel must be open for update, append, or output to the disk file.

Printer

A PRINT statement for a printer also requires a previous OPEN statement. Some of the characters may not be printed exactly as they are displayed on the screen. Different types of printers have different character sets, so the actual results depend on the type of printer being used.

Display

The use of a PRINT statement in the graphics mode is complicated but not difficult. In graphics modes 1 and 2, the PRINT statement displays characters on the screen. In graphics modes 3 through 8, the PRINT statement displays a picture element on the screen.

A PRINT statement can be used in a graphics mode if an OPEN statement has been executed for output to device "S:". Also, a PRINT statement can be used in a graphics mode if a GRAPHICS statement has been executed.

The PRINT statement must include an I/O channel number if a corresponding OPEN statement has been executed. The PRINT statement must include I/O channel number 6 if a GRAPHICS statement was executed. When a PRINT statement is used with graphics, the I/O channel number should be followed by a semicolon instead of a comma.

In graphics modes 1 and 2, the characters are displayed as they appear in Table 9-4. There are two sets of characters available by executing appropriate POKE statements. Each of these characters can be displayed in any one of four colors.

In Table 9-4, each character has four numbers associated with it. These four numbers correspond to the four color registers. In order to display one of the characters from Table 9-4 with a PRINT statement, the PRINT statement must include the character from Table 9-4 that has the same number as the desired character from Table 9-4.

EXAMPLE

```
10 GRAPHICS 2
20 PRINT #6; "ATARI"
30 PRINT #6; "atari"
40 END
```

The previous example contains a program that uses two PRINT statements in graphics mode 2. At line 20, the PRINT statement indicates I/O channel number 6, which is used for graphics. To determine which characters are to be printed, it is necessary to consult Table 9-7 first. The upper case letters "ATARI" are represented on Table 9-7 by the values 65, 84, 65, 82, and 73. The characters that correspond to these values, on Table 9-4, are the characters that will be displayed. These values indicate the characters "ATARI" in color register 0 from Table 9-4.

Similarly, the second PRINT statement has the lower case letters "atari". These characters have the values 97, 116, 97, 114, and 105. On Table 9-4, these values correspond to the upper case letters "ATARI" in color register 1. As a result of this program, the message "ATARI" appears in orange and in light green.

In graphics modes 3 through 8, a PRINT statement illuminates one picture element for each character in the PRINT statement. The color of the picture element is derived from the ASCII code of the character. In the four-color graphics modes (3, 5, and 7), the ASCII code is reduced modulo 4 to a number from 0 to 3. This value corresponds to the color value of the picture element. In the two-color graphics modes (4, 6, and 8), the ASCII code is reduced modulo 2 to indicate color value 0 or 1.

Atari 850 Interface

A PRINT statement for a serial port of an Atari 850 Interface module must be preceded by an appropriate OPEN statement. Also, the interface module must be ready to operate. The interface module will not operate unless it was turned on before the computer console was turned on.

PTRIG

The PTRIG function returns the value of 0 if the specified paddle controller button is depressed. A 1 is returned if the button is released.

CONFIGURATION

$X = \text{PTRIG}(a)$

The value of the argument (a) indicates the number of the paddle. A pair of controllers can be plugged into each of the controller jacks on the computer. The first jack accepts paddles 0 and 1. The second jack accepts paddles 2 and 3, etc. A total of 8 paddle controllers can be used at one time.

EXAMPLE

IF PTRIG(3) = 0 THEN END

The statement in the preceding example ends the program if the button on paddle 3 is being pressed.

PUT (PU.)

The PUT statement is used to send one byte to an output device. One byte represents an integer between 0 and 255.

CONFIGURATION

PUT #a, b

The first argument of a PUT statement is the I/O channel number. The second argument is the value that is sent to an output device. If either of the arguments are not integers, they are rounded off.

An OPEN statement must precede the PUT statement except when the PUT statement is used with graphics displays. The first argument of the PUT statement must correspond to the I/O

channel number in the OPEN statement. When a PUT statement is used for the display following a GRAPHICS statement, the I/O channel number must be 6.

The I/O channel must be open for output to an appropriate output device.

The second argument can be any non-negative value, but the value that is sent to the output device will always be an integer between 0 and 255. Larger values are reduced modulo 256.

With the Program Recorder, an OPEN statement is needed to open an I/O channel for device "C:". When the Program Recorder is used for short gaps between records, the tape keeps moving until the I/O channel is closed. As a result, the program must keep up with the tape or the information will not be recorded properly.

EXAMPLE

```
10 OPEN #1, 8, 0, "C:"  
20 FOR I = 1 TO 100  
30 X = INT(RND(9) * 100)  
40 PUT #1, X  
50 NEXT I  
60 END
```

The previous example contains a program that records 100 random numbers. Line 10 is an OPEN statement that opens I/O channel number 1 for output to the Program Recorder. Line 20 is a FOR statement that begins a FOR/NEXT loop that is repeated 100 times. Line 30 sets X equal to a random integer between 0 and 100.

Line 40 contains a PUT statement that sends the value of the variable X to the output device on I/O channel number 1. Since the I/O channel is open for output to the Program Recorder, the values of the variable X are recorded on the cassette tape. When the FOR/NEXT loop has been executed 100 times, the END statement at line 60 closes the I/O channel and ends the program.

The same format of the PUT statement is used with the Atari disk drive. The I/O channel for the disk file must be opened for an appropriate output option. Only a GET statement can be used to recover the values that were recorded with a PUT statement.

When the display is used as an output device (S: or E:), a PUT statement is used to place one character or illuminate one picture element on the screen. The PUT statement causes the output to appear at the current position of the cursor.

In the text modes, the value of the PUT statement corresponds to the COLOR value of each character (Tables 9-4 and 9-7). As a result, a POSITION and PUT statement have the same result as a COLOR and PLOT statement.

EXAMPLE

```
10 GRAPHICS 2
20 POSITION 9,3
30 PUT #6,65
```

The previous example contains a program that uses a PUT statement in graphics mode 2. Line 20 positions the cursor near the center of the screen. The graphics modes always use I/O channel number 6. As a result, the PUT statement at Line 30 displays the character "A" at the current cursor position. The characters that correspond to the PUT statement values are listed in Tables 9-4 and 9-7.

When a PUT statement sends the value 125 to the screen, the display is cleared. Also, when the value 155 is sent to the screen, the cursor returns to the beginning of the next line.

In the four-color graphics modes (3, 5, and 7), the value that a PUT statement sends to the screen is reduced modulo 4 to a value between 0 and 3. The PUT statement illuminates the picture element at the current position of the cursor.

The color of the picture element is determined by the value between 0 and 3 in the same way that a COLOR statement value determines the color.

In the two-color graphics modes (4, 6, and 8), the value of the PUT statement is reduced modulo 2 to the numbers 0 or 1. The color of the picture element is determined by the values 0 and 1 in the same way that a COLOR statement determines the color.

A PUT statement can also be used to send output to a printer. The printer must be ready to operate when the corresponding OPEN statement is executed.

PUT can also be used to send data to an open RS-232 serial port on the Atari 850 Interface Module.

RAD

The RAD statement causes the trigonometric functions to be performed in radians.

CONFIGURATION

RAD

EXAMPLE

RAD

The trigonometric functions are performed in radians until a DEG statement is executed. Also, radians are used following a NEW or RUN statement or following a System Reset.

READ (REA.)

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

READ

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 specified in DATA. A numeric variable can only be assigned a numeric value. However, a string variable can accept any type of characters or none at all.

String variables must be correctly dimensioned before the READ statement for that variable is executed.

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

```
10 DIM X$(10)
20 READ X,X$
30 PRINT X$,X
40 END
50 DATA 12, JONES
RUN
JONES    12
```

The preceding example contains a program that has a READ statement. First, the string variable X\$ is dimensioned. Next, 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 (R. or .)

A REM statement is used to insert comments in a program. The REM statement is ignored by the Atari 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 (RES.)

A RESTORE statement is used to move the data pointer.

CONFIGURATION

RESTORE [LN]

The data in a program is read in order, starting with the first DATA statement item. In order to reread a section of data, a RESTORE statement is necessary.

When a RESTORE statement is executed without an argument, the next READ statement will assign to its first variable the first data value that appears in the program.

When a RESTORE statement is executed with an argument, the next READ statement will assign to its first variable the first data value that appears at the line number specified by the argument.

EXAMPLE

RESTORE 100

The previous example contains a statement that moves the data pointer to the DATA statement at line 100. If line 100 is not a DATA statement, the data pointer is moved to the next DATA statement after line 100.

RETURN (RET.)

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 the program control to return to the statement following the most recently executed GOSUB or ON/GOSUB statement.

EXAMPLE

RETURN

When a POP statement is executed before a RETURN statement, the most recent GOSUB statement is ignored, and the program control is branched to the next most recent GOSUB statement.

RND

The RND function is used to generate random numbers.

CONFIGURATION

$X = \text{RND}(a)$

The argument of a RND statement has no effect on the results, but it is necessary. The value of the random number is less than 1 and greater than or equal to zero.

EXAMPLE

```
X = INT(RND(9) * 100)
```

The previous example contains a statement that generates random integers between 0 and 99 inclusive.

RUN (RU.)

The RUN statement is used to execute the program that is currently in the computer's memory. A RUN statement is also used to load and execute a program from an input device.

CONFIGURATION

```
RUN ["device:filespec"]
```

A RUN statement closes the I/O channels and turns off the sound voices before executing or loading the program.

When a RUN statement is used with an input device, the contents of the computer's memory are erased before the program is loaded. Only BASIC programs that were recorded with the SAVE statement can be loaded and executed with a RUN statement.

The Program Recorder is activated with a RUN "C:" statement. The tone sounds once to remind the operator to position the tape and press the Play lever on the Program Recorder followed by Return on the computer's keyboard.

A RUN statement can load and execute a program from a disk file if the disk operating system has been booted. An error results if the specified file does not exist.

EXAMPLES

```
RUN "C:"  
RUN "D2:JONES.BAS"
```

SAVE

The SAVE command is used to send a BASIC program in RAM to an output device.

CONFIGURATION

SAVE *device*

where *device* is a device name such as the program recorder (C:) or disk drive (D:). In the case of the disk drive, a filename may be specified with the device name. The program will be saved under the filename specified.

Files saved via SAVE are transferred in tokenized format. These files can only be subsequently loaded using LOAD or RUN. CLOAD and ENTER will not load a program saved with SAVE.

SAVE With The Program Recorder

The SAVE C: command is used to transfer a program to the program recorder. When SAVE C: is executed, the Atari's speaker will sound twice to indicate that the tape is to be positioned correctly to receive the file. Once the tape has been positioned, press the Record and Play buttons on the recorder. Then, press any key on the Atari's keyboard. The program will then be transferred from RAM to the program recorder.

SAVE With The Disk Drive

Before SAVE can be used to transfer a program to the disk drive, DOS must have first been booted. An error will result if an attempt is made to execute SAVE when DOS has not been booted. If a file with the same filename as the file specified with SAVE already exists on the diskette to which the program is being transferred, the file being transferred will replace the file on diskette with the same name.

SETCOLOR (SE.)

The SETCOLOR statement is used to assign a color and luminance value to the color register specified.

CONFIGURATION

SETCOLOR register #, color, luminance

The color register must range from 0 to 4 inclusive. The color must range from 0 to 15 inclusive. These values and their corresponding colors are listed in Table 9-3. The luminance can range from 0 (darkest) to 14 (brightest).

Each of the 5 color registers has a default color and luminance value. These default values are listed in Table 9-2.

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
RUN
1
```

SIN

The SIN function returns the sine of the angle specified as its argument. The argument will be assumed in radians unless a DEG statement precedes the SIN statement.

CONFIGURATION

X = SIN (a)

EXAMPLE

```
10 DEG
20 X = SIN (90)
30 PRINT X
RUN
1
```

SOUND

The SOUND statement is used to output sound via the television set or monitor's speaker.

The SOUND statement is used with the following configuration.

SOUND *voice, pitch, distortion, volume*

Together these four arguments determine the sound produced. *voice* sets one of four voices available with the Atari. These are numbered from 0 to 3. These four voices are independent of each other. In other words, as many as four voices can be sounded at the same time.

pitch sets the pitch of the sound produced by the SOUND statement. The pitch can range from 0 to 255. The highest pitch begins at 0 and the lowest at 255.

The SOUND statement can produce either pure or distorted tones. *distortion* can range between 0 and 15. A *distortion* value of 10 or 14 will produce a pure tone. Any of the other even distortion values (0, 2, 4, 6, 8, 10, and 12) will generate a different amount of noise into the tone produced. The amount of this noise will depend upon the distortion and pitch values specified.

The odd numbered *distortion* values (1, 3, 5, 7, 9, 11, 13) cause the *voice* indicated in the SOUND statement to be silenced. If the *voice* is on, an odd-numbered *distortion* value will result in its being shut off.

The *volume* controls the loudness of the voice indicated in SOUND. *volume* ranges from 0 (no sound) to 15 (highest volume).

An Atari BASIC statement with a volume of 0 will turn off the sound. Sound can also be turned off by executing an END, RUN, NEW, DOS, CSAVE, or CLOAD. If the System Reset key is pressed, sound will be turned off. However, if the Break key is pressed, sound will not be turned off.

SQR

SQR returns the square root of its argument.

CONFIGURATION

SQR (a)

EXAMPLE

```
10 X = 49
20 PRINT SQR (X)
RUN
7
```

STATUS

STATUS returns a code which identifies the last input/output operation undertaken on the channel specified.

CONFIGURATION

STATUS #channel, X

The status code will be returned via the numeric variable indicated. The status codes are listed in Table 5-3.

EXAMPLE

```
100 STATUS #5, ST4
200 PRINT ST4
RUN
130
```

In the preceding example, the status code for the last input/output activity undertaken on the device opened as channel 5 is displayed.

Table 5-3. STATUS Code Values

STATUS Code	Reference
1	Operation completed with no problem. Approaching end of file, Next READ receives last data in file. Reference Error Messages 128-171 in Appendix A.
3	
128-171	

STICK

The STICK function returns the position of the joystick indicated as its argument.

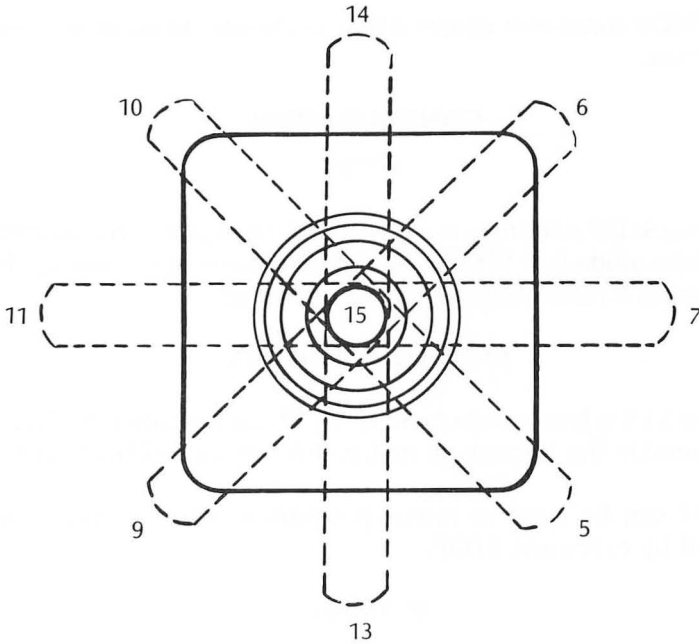
CONFIGURATION

STICK (a)

"a" indicates the joystick number (0-3). The value returned can range from 0 to 15 and corresponds to the positions indicated in Illustration 5-1.

EXAMPLE

IF STICK (1) = 7 THEN GOTO 700

Illustration 5-1. STICK Joystick Positions

STRIG

The STRIG function returns a value of 0 if the specified joystick's button is depressed. A 1 is returned if the button is released.

CONFIGURATION

STRIG (a)

"a" indicates the joystick number (0-3).

EXAMPLE

```
100 IF STRIG (2) = 0 THEN GOTO 700
```

STOP

The STOP statement causes a halt in the execution of a BASIC program.

CONFIGURATION

STOP

When a STOP statement is executed, the computer will return to graphics mode 0. If STOP is executed in the program mode, the following screen message will be displayed;

STOPPED AT LINE XXX

where XXX is line number where STOP was executed. If STOP is executed in the immediate mode, this message will not appear.

CONT can be used to rescue program execution after it was halted by executing STOP.

EXAMPLE

```
100 INPUT A
105 IF SGN (A) = -1 THEN 150
110 B = SQR (A)
120 IF SGN (B) <> 1 THEN STOP
130 PRINT B
140 GOTO 100
150 END
```

In the preceding example, if a value of 0 is input for A in line 100, program execution will stop and the following message will be displayed.

STOPPED AT LINE 120

By entering CONT, program execution will resume with line 130.

STR\$

STR\$ returns the string representation of its argument.

CONFIGURAITON

a\$ = STR\$(a)

In the following example, A\$ would consist of the string "40". In this case, "40" is a string--not a number. In other words, "40" (in its string equivalent) would not be used in calculations.

EXAMPLE

```
050 DIM A$(50)
150 A$ = STR$(40)
200 PRINT A$
RUN
40
```

TRAP

The TRAP statement causes program execution to branch to the line number indicated when an error is encountered.

CONFIGURATION

TRAP LN

TRAP must have been executed prior to the occurrence of the error. Otherwise, a branch to the indicated program line will not take place.

TRAP will invalidate the Atari's automatic error handling routine which halts program execution.

EXAMPLE

```

100 TRAP 700
200 INPUT A
300 IF A = 0 THEN 999
400 PRINT A
500 GOTO 200
700 PRINT PEEK (195)
800 PRINT 256 * PEEK (187) + PEEK (186)
999 END
RUN
?A
8
200
READY

```

In the preceding example, the TRAP statement in line 100 will cause the program to branch to line 700 if an error is encountered. In line 700, the error code is displayed. (Address 195 is used to store the error code.). In line 800, the line number where the error occurred is displayed. The following expression,

$$256 * \text{PEEK}(187) + \text{PEEK}(186)$$

returns the line number where the error occurred.

In our example, the data input in response to the INPUT statement in line 200 was string. Since a numeric variable was specified in line 200, error code 8 was generated. This was displayed along with the line number where the error occurred (200).

USR

USR is used to branch program control to a machine language program.

CONFIGURATION

USR(address[, argument ...])

The address indicated is that of the machine language subroutine to be branched to. Function arguments between 0 and 65535 can be optionally included with the USR command as indicated in the Configuration.

Beginning with the last argument, each argument is evaluated and converted to a 2-byte hexadecimal integer. This integer is placed on the hardware stack, and a count of the USR arguments is also pushed on the stack. The hardware stack configuration is depicted in Illustration 5-2.

Returning To BASIC

When BASIC executes a USR function, the BASIC program's current location is pushed onto the hardware stack (see Illustration 5-2). The machine language program can return to BASIC by executing the assembly language RTS instruction. RTS will pull the return location within the BASIC program from the hardware stack.

However, before RTS can be used to pull the return location off the stack, all data on the stack related to function arguments must have been pulled off the stack. This includes both the arguments themselves as well as the argument count. Even if there are not arguments, the machine language program must pull the argument count off the stack before returning to the BASIC program.

VAL

The VAL function converts its string argument to a numeric value. The first character of the string argument must be a numeric character. Otherwise, an error will occur. The numeric characters in the string argument will be converted to their numeric equivalents until a non-numeric string character is encountered.

CONFIGURATION

VAL (a\$)

Illustration 5-2. USR Hardware Stack

Top of Stack

USR Argument Count
First USR Argument
Second USR Argument
⋮
Final USR Argument
BASIC Program's Return Address
Stack Contents Prior to USR

Bottom of Stack

EXAMPLE

```

050 DIM A$(50)
100 A$ = "57A72B"
200 PRINT VAL(A$)
300 PRINT VAL(A$) + 2
RUN
57
59

```

XIO

The XIO statement is a generalized input/output statement which can perform a wide range of input and output operations. These operations are summarized in Table 5-4.

CONFIGURATION

XIO command, #channel, a, b, device

The command value (as specified in Table 5-4) indicates the operation to be performed. The channel specified must have been previously opened for input or output (with the exception of XIO 3).

The numeric expressions (a, b) are not always used by XIO, however, they must always be present as parameters. The applicable numeric expression values are given in Tables 5-4, 5-5, 5-6, and 5-7.

The final parameter, device, specifies the device to be used for the input/output operation.

Table 5-4. XIO Command Summary

Operation	Command	DOS or BASIC Counterpart	Numeric Exp1	Numeric Exp2
General I/O Operations:				
Open a channel	3	OPEN	See Table 5-1	0-8
Read a line	5	INPUT	0	0
Get a character	7	GET	0	0
Write a line	9	PRINT	0	0
Put a character	11	PUT	0	0
Close channel	12	CLOSE	0	0
Status of channel	STATUS	0	0	
Screen Graphics:				
Draw a line	17	DRAWTO	0	0
Fill an area	18	None	0	0
Disk*:				
Rename a file	32	DOS Menu E	0	0
Delete a file	33	DOS Menu D	0	0
Lock a file	35	DOS Menu F	0	0
Unlock a file	36	DOS Menu G	0	0
Move file pointer	37	POINT	0	0
Find file pointer	38	NOTE	0	0
Format diskette	254	DOS Menu I	0	0

*DOS must have been booted.

RS-232 Serial Port:				
Output Port of a Block	32	None	0	0
Control DTR, RTS, XMT	34	None	See Table 5-5	0
Baud rate, word size, stop bits, & ready monitoring	36	None	See Table 5-6	See Table 5-6
Translation mode	38	None	See Table 5-7	ASCII Code
Concurrent mode	40	None	0	0

Table 5-5. Numeric Expression 1 Values for XIO 34

Function*	DTR	RTS	XMT
No change	0	0	0
Turn Off (XMT to 0)	128	32	2
Turn On (XMT to 1)	192	48	3

*Add value for DTR, RTS, & XMT to obtain Numeric Expression 1

Example Values of Numeric Expression 1	DTR	RTS	XMT
162	Off	Off	0
163	Off	Off	1
178	Off	On	0
179	Off	On	1
226	On	Off	0
227	On	Off	1
242	On	On	0
243	On	On	1

**Table 5-6. Numeric Expression 1 and 2 Values
For XIO 36**

Numeric Expression 1 Value*					
Stop Bits	Value	Word Size	Value	Baud Rate	Value
1	0	8 bits	0	300	0
2	128	7 bits	16	45.5	1
		6 bits	32	50	2
		5 bits	48	56.875	3
				75	4
				110	5
				134.5	6
				150	7
				300	8
				600	9
				1200	10
				1800	11
				2400	12
				4800	13
				9600	14
				9600	15

*Add value from each column to determine Numeric Expression 1.

Numeric Expression 2 Value

DSR	CTS	CRX	Value
No	No	No	0
No	No	Yes	1
No	Yes	No	2
No	Yes	Yes	3
Yes	No	No	4
Yes	No	Yes	5
Yes	Yes	No	6
Yes	Yes	Yes	7

Table 5-7. Numeric Expression 1 Value for XIO 38

Numeric Expression 1*

Line Feed		Translate Atari ASCII to ASCII		Input Parity		Output Parity	
Append	Value	Mode	Value	Mode	Value	Mode	Value
No Yes**	0	Light	0	Disregard	0	No change	0
	64	Heavy	16	Odd	4	Odd	1
		None	32	Even	8	Even	2
				Disregard	12	Bit On	3

*Add one value from each column to determine Numeric Expression 1.

**The line feed character is appended after a carriage return (EOL).

EXAMPLE

XIO Example Program

```
100 GRAPHICS 5
200 COLOR 1
300 PLOT 50,20
400 DRAWTO 50,10
500 DRAWTO 10,10
600 POSITION 20,20
700 POKE 765,1
800 XIO 18,#6,0,0,"S:"
```

The preceding example illustrates the use of the XIO command to fill an area in graphics. The command, 18, specifies the graphics fill area action. Channel #6 is the graphics channel. The numeric parameters are both specified as 0, and the device is the screen (S:).

CHAPTER 6.

ATARI 410 PROGRAM RECORDER

Introduction

The Atari 410 Program Recorder is used for storing BASIC programs or data on cassette tape. BASIC programs or data can be transferred from RAM onto cassette via any one of several Atari BASIC statements.

The process of transferring a program from RAM onto cassette tape (or any other storage device) is known as **saving** that program. Once a program has been saved, it can later be transferred back from the storage device into RAM. This process is known as **loading**.

Data can also be transferred back and forth between RAM and cassette tape. The process of sending data to cassette tape is known as **writing** the data. The retrieval of that data from cassette tape back into RAM is known as **reading** the data.

In this chapter, we will discuss the BASIC statements used to read and write data and to save and load programs. However, first we will discuss the concepts of data and program storage.

Data Files-Files, Records, & Fields

Data files can be visualized as being organized as **files**, **records**, or **fields**.

If we visualized the Atari 410 Program Recorder as a filing cabinet, a data file would be analogous to one file within that filing cabinet. For instance, if you kept a file filled with slips of paper containing the names and addresses of all of your cousins, that physical file would be analogous to a computer's data file.

Your data file could contain any number of slips of paper--depending upon how many cousins you had. Each slip of paper containing the name and address of one of your cousins would be analogous to a **record** with a data file.

Each individual data item within a record is known as a **field**. In our example, the name of each cousin might be considered a field as well as the street address, city, state, zip code, and telephone number.

Program Files

Programs are also stored as files. However, unlike data files, program files are not divided into records and fields.

We will discuss loading and saving program files in the following several sections. The reading and writing of information to data files will be discussed later in this chapter.

Saving Programs on the Atari 410

Atari BASIC contains three statements that are used to store programs on cassette tape. These are:

CSAVE
SAVE
LIST

Each of these three statements has a corresponding Atari BASIC statement which is used to load a program into memory from the cassette tape. These are:

CLOAD
LOAD
ENTER

The CSAVE statement is used only for saving programs on cassette. LIST and SAVE can be used to send a program to devices other than the Atari 410 Program Recorder. LIST and SAVE must

be used with the Program Recorder's device name (C:) as shown below,

```
LIST "C:"  
SAVE "C:"
```

to save programs on the Program Recorder.

Either the CSAVE or SAVE "C:" statements will save the complete program when executed. The LIST "C:" statement can be used to save either all or part of a program. The following LIST statement would save line numbers between 500 and 1000 to the 410 Program Recorder when executed.

```
LIST "C:" 500, 1000
```

When either the CSAVE, LIST "C:", or SAVE "C:" statements are executed, the Atari's speaker will sound twice. This is a signal for the operator to place a cassette tape in the 410 Program Recorder. The tape should then be forwarded to the position when recording is to begin. The rewind and fast forward keys can be used to position the tape.

Once the tape is in the proper position, press the record and play keys. When the tape is ready, press Return on the Atari keyboard and the recording process will begin.

By turning up the sound on your television set or monitor, you can actually hear the recording process. A high pitched tone will be sounded followed by a number of short eruptions of sound. Each sound indicates that a block of program information has been saved on the cassette. When the program has been recorded, the sounds will stop and the tape will stop. The user should then press the stop key on the Atari 410 Program Recorder.

Program Recording Formats

Each of the three Atari BASIC statements used to record programs do so using a different format. These different formats are not discernable to the user. However, the user must keep in

mind the format used to save an Atari program if he wishes to successfully load that program.

The CSAVE and SAVE statements record programs in **tokenized** format. In this format, Atari BASIC keywords are abbreviated with one character tokens. The computer automatically encodes the keywords as tokens.

Although both the SAVE and CSAVE statements record programs in tokenized format, differences remain in the exact format used. Programs are recorded in groups of data known as **blocks**. The difference between SAVE and CSAVE lies in the amount of space allowed to remain between the blocks of data.

The CSAVE statement allows less space between these blocks of data than does the SAVE statement. Therefore, saving or loading a program with CSAVE and CLOAD will be accomplished in less time than with SAVE "C:" and LOAD "C:".

CLOAD will load programs saved with either the CSAVE or SAVE statements, while the LOAD statement will only load those programs recorded with a SAVE statement.

The LIST statement saves programs in Atari ASCII format. An ASCII code is saved for every character in the Atari BASIC program. Keywords are not abbreviated as tokens. The ENTER statement must be used to load programs saved with the LIST statement.

Loading a Program on the Atari 410

As previously mentioned, the CLOAD statement is used to load programs from cassette tape into RAM that had been previously saved with the SAVE or CSAVE statements. Likewise, the LOAD statement is used to load those programs previously saved with the SAVE statement, and the ENTER statement is used to load programs saved with the LIST statement.

The reason why certain statements must be used to load files saved with corresponding statements lies in the format in which the program was recorded. The LIST statement saves a BASIC program file in Atari ASCII format, while SAVE and CSAVE

transfer BASIC programs in tokenized format.

The ENTER statement can only load a BASIC file stored in ASCII format. ENTER will not load BASIC files stored in tokenized format.

LOAD and CLOAD can only load BASIC files stored in tokenized format. CLOAD will load files saved with either SAVE or CSAVE, but neither will load files saved with LIST.

LOAD can only load files saved with the SAVE statement in tokenized format. LOAD cannot be used to load files saved with CSAVE even though CSAVE stores BASIC files in tokenized format, because CSAVE uses a timing pattern which is incompatible with LOAD. LOAD cannot be used with BASIC programs saved with the LIST statement as these were saved in ASCII format.

Both ENTER and LOAD can be used to transfer a program from a device other than the Atari 410 program recorder. The CLOAD statement can only be used to load a program from the Atari 410 into RAM.

When either LOAD or CLOAD are used to load a program, a NEW statement will be automatically executed before the program is loaded. This causes any existing programs or variables to be erased from memory.

If the ENTER statement is used to load a program from the Atari 410, any existing program lines will not be erased from memory. ENTER adds the program lines from the cassette file to any existing program lines in memory. If a program line in the program being transferred from cassette has the same line number as a program line in memory, the program line being transferred from cassette will replace the program line in memory.

Also, when ENTER is used to load a program, any variables in the **variable name table** (VNT) will not be erased. Any new variable names encountered in the program being loaded will be added to the existing variable names in the VNT.

The variable name table is a table kept by Atari BASIC of all variable and array names used in a program regardless of whether the program was entered in the immediate or the program mode.

When the CSAVE or SAVE statements are used to save a BASIC program, the variable name table is recorded with the program lines. When CLOAD and LOAD are executed to load the program lines back into RAM, the variable name table will also be loaded and will replace any existing variable name table in RAM.

Conversely, when the LIST statement is executed to save a BASIC program, the variable name table will not be saved. Therefore, when the ENTER statement is executed to load the program saved by LIST back into memory, no variable name table will be loaded and the variable name table currently held in RAM will remain. When the BASIC program loaded with ENTER is executed, any variable and array names used in that program will be added to the variable name table.

If programs are continually saved and loaded with LIST and ENTER statements, the variable name table may eventually become overcrowded with unused variable and array names. It may become necessary to clear the variable name table.

This can be accomplished by first saving the existing program in RAM using the LIST statement. Next, by executing the NEW statement, the variable name table (as well as the existing program in RAM) will be erased. The program can then be loaded back into RAM using the ENTER statement. As the program is executed, variable and array names will be added to the variable name table.

When either of the following statements,

```
CLOAD  
ENTER "C:"  
LOAD "C:"
```

are executed, the following series of events will occur.

1. The computer's built-in speaker will sound one time. This is a signal to the operator to place the cassette containing the program to be loaded in the Atari 410 Program Recorder.
2. Use the rewind and fast forward keys to position the tape to the area on the tape near the beginning of the program. It is a good practice to save a program at the beginning point of a tape as it is then easy to locate.

For programs not recorded at the beginning of a tape, the 410's tape counter can be used to locate a program's beginning position.

3. Once the beginning position of the program on the tape is located, the play button on the 410 Program Recorder should be pressed. Also, the Return key on the Atari computer's keyboard must be pressed to signal the computer that the cassette is ready.
4. The cassette tape will then begin moving as the program is loaded into memory. By turning up the volume on your monitor or television set, you can actually hear the program being loaded. You will hear short eruptions of sound followed by long periods of silence. Each sound eruption is emitted as a block of data is loaded from the cassette.

When the sound eruptions stop, the tape will stop as well. The entire program has now been loaded into RAM. The stop key on the 410 Program Recorder should then be pressed to stop the tape.

RUN "C:" Statement

The RUN "C:" statement is a variation of the RUN statement which allows the user to both load and run a program from the 410 Program Recorder in a single step.

The RUN "C:" statement is in fact a combination of the LOAD "C:" and RUN statements. Therefore, RUN "C:" can only be used to load programs saved with the SAVE "C:" statement.

The RUN "C:" statement can be used in the program mode as well as the immediate mode to load and execute a program. For example, suppose the following program was input and saved on cassette:

```
NEW  
READY  
100 PRINT "ONE, TWO, THREE"  
200 PRINT "PRESS RETURN TO LOAD"  
300 RUN "C:"  
SAVE "C:"  
READY  
■
```

A second program could then be input and saved as shown in the following example:

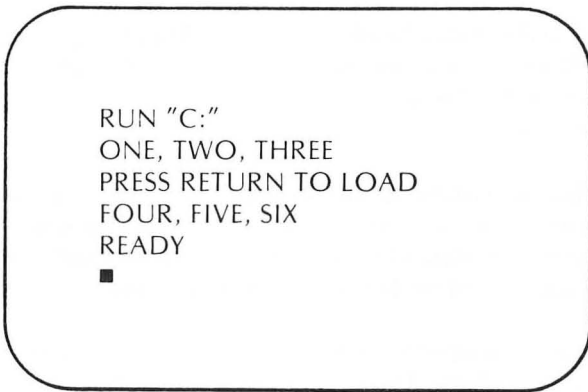
```
NEW  
READY  
100 PRINT "FOUR, FIVE, SIX"  
SAVE "C:"  
READY  
■
```

The cassette tape now contains two consecutive programs. By rewinding the tape, entering "NEW", and executing the RUN "C:" statement, the first program will be loaded and executed.

When line 300 is executed, the RUN "C:" statement will cause

the Atari's speaker to beep. When the operator presses the Return key on the Atari's keyboard, the second program will be automatically loaded and executed.

Notice the inclusion of the prompt statement in line 200 of the first program to remind the operator to press the Return key. The output from the loading and execution of our examples should appear as in the following:



This procedure of loading the program from another is known as **chaining**. When chaining programs using the RUN statement, remember that a NEW statement is automatically executed causing the existing program in memory as well as the variable name table to be erased. This prevents a program from using the same variable values that were used in a previous program from which it was chained.

Reading and Writing Data

Atari BASIC uses the PRINT# and PUT statements to write data to the 410 Program Recorder. The INPUT# and GET statements are used to read data stored on cassette back into RAM.

When data is being transferred between RAM and the 410 Program Recorder, it is transferred in blocks of 128 characters. A portion of memory is reserved to hold one 128 character block of data that is to be read from or written to the 410 Program

Recorder. This area of memory is known as the **cassette buffer**.

As mentioned earlier in this chapter, data is organized into files, records, and fields. The notion of files, records, and fields relate to a programmer's conception of how a data file is organized.

In actuality, a cassette data file physically consists of three separate parts: the leader, data blocks, and an end-of-file record.

The 20 second data file leader allows the Atari computer and 410 Program Recorder to synchronize their timing so that data may be transferred. The actual data is stored in blocks of 128 characters each.

The last block (known as the end-of-file block) can consist of fewer than 128 characters of data. This block holds the final characters of the data file (i.e. Those characters still remaining after the last complete block has been utilized.).

Records can consist of more than one block, less than one block, or exactly one block. The computer will assign records to data blocks automatically. The programmer need not concern himself with this process.

Opening Data Files

Before information can be read from or written to a data file, that file must first be opened. This is accomplished with the OPEN statement.

The data file can be read from or written to as long as it is open. To prevent access to the file, it must be closed. This is accomplished with the CLOSE statement.

The OPEN statement uses the following configuration:

OPEN #*channel*, *task*, *value*, *device*

OPEN can be used with external devices other than the 410 Program Recorder including the disk, screen, keyboard, printer,

and RS-232 port.

The following is an example of an OPEN statement used to open a data file for access by the 410 Program Recorder.

```
OPEN #1, 4, 0, "C:"
```

The first parameter in the OPEN statement indicates the *channel*. Before an external device can be accessed either for input or output, a channel to the device must have been opened.

The OPEN statement assigns a channel to an external device. Once a channel is assigned, the device can subsequently be accessed via its channel number. Only one channel can be open at any one time. If an OPEN channel attempts to open a channel which is already open, an error will occur.

The second parameter in the OPEN statement indicates the activity for which the channel is being opened. For the 410 Program Recorder, a value of 4 indicates the channel is being opened to read data while a value of 8 indicates the channel is being opened to write data. A cassette file cannot be simultaneously open for both input and output.

The third parameter in the OPEN statement is 0 for standard data files. The final parameter, *device*, must be specified as "C:" for input or output to the 410 Program Recorder.

When a cassette data file is opened for input or output, the Atari's speaker will be sounded once for input or twice for output to signal the operator to properly position the tape. Once the tape has been properly positioned, the user should press the 410's play button, and then press any key on the Atari.

If the file was opened for output, the computer will write the leader. If the file is opened for input, it will read the leader. This process requires approximately 20 seconds.

Once the leader has been read or written, the program must read or write data to the open file. If the file was opened for output, data must be written to the file. If the program fails to

immediately write data to a newly opened file, an error may result if an attempt is later made to read the file.

If the file was opened for input, the program must read data from the file with a GET or INPUT# statement. If the program fails to read data from the open file, an error may occur if the program subsequently attempts to read data in the program.

Closing Data Files

Once an open file has been accessed, it is important to close that file so that the file's channel can be assigned elsewhere.

Also, if the channel is open for output, closing it causes any remaining data in the cassette buffer to be output followed by an end-of-file record. If the open file is not closed, any remaining data in the cassette buffer may be lost.

The following CLOSE statement,

CLOSE #4

will close channel 4.

All open files are closed automatically when an END or RUN statement is executed, or when a program's last statement is executed (only in the program mode).

Writing to a Data File

As mentioned previously, the PRINT# and PUT statements can be used to send data from RAM to the cassette data file. Both PRINT# and PUT specify an open channel as their first parameter. The data being output will be sent via the channel indicated. That channel must be open.

The PRINT# statement uses the following configuration when used to output data to a cassette file.

PRINT #*channel*; *expression*...

channel indicates the channel to be used to send the output. Semicolons should be used rather than commas to separate the *channel* from the first *expression*, as well as to separate any subsequent optional *expressions*. The use of commas is allowed, but this practice would cause additional blank spaces to be inserted in the file. Each separate *expression* should end with an EOL character. This can be accomplished by using one PRINT# statement for each *expression* or by using CHR\$(155).

The *expressions* consist of one or more string or numeric values to be output. These values are output as ASCII values.

Always be sure that a PRINT# statement being used to output data to a cassette file outputs an end-of-line (EOL) character after each *expression*. The PRINT# statement sends data to the cassette buffer where it is stored until it is filled. The entire block of data (128 bytes) is then sent to the Atari 410.

If an EOL character is output at the end of the PRINT# statement, the data in the cassette buffer will be output to the cassette file, regardless of whether the buffer is full or not. In these situations, the 128th character in the buffer will contain the actual number of bytes in the buffer. This value is stored in the hex form.

PRINT# statements automatically output an EOL character after outputting the *expressions* unless a comma or semicolon is placed at the end of the *expression* list. For this reason, never end a PRINT# statement with a comma or semicolon, when it is being used to output data to a cassette file. An example of the use of PRINT# is given below.

```
400 OPEN #2, 8, 0, "C:"
500 PRINT#2; "JOHN"
550 PRINT#2; "JACK"
600 CLOSE #2
```

The PUT statement can be used to output a single numeric value to the cassette file via an open channel. PUT uses the following configuration:

PUT #*channel*, *numeric expression*

As with PRINT#, the *channel* specified must be open for output. The value given in *numeric expression* is output to the data file. The value output will lie between 0 and 255. If the value is not an integer, it will be rounded to the nearest integer.

If the value specified does not lie between 0 and 255, it will be output modulo 256. In other words, 256 would be sent as 0, 257 as 1, 258 as 2, 259 as 3, etc.

An example of a PUT statement is given below.

```
150 OPEN #2, 8, 0, "C:"
160 PUT #2, 123
170 CLOSE #2
```

Reading from Data Files

The INPUT# and GET statements are used to read data from cassette data files. INPUT# will accept data from the cassette data file, interpret that data, and assign the data to the variable or variables names in its parameter list.

The following configuration is used with INPUT#:

$$\text{INPUT \#channel } \left\{ \begin{array}{l} ' \\ ; \end{array} \right\} \text{ variable, ...}$$

The *channel* must have been opened previously for input. The *variables* will accept the data values input.

INPUT# will retrieve data from the input device specified. This data will consist of ASCII characters followed by the ASCII end-of-line character. The EOL character will end input to the *variable* specified.

INPUT# will interpret the data being read as either numeric or string--depending on the type of *variables* used as parameters. When a numeric variable is specified, the data being input will be interpreted as numeric data.

The data read via INPUT# will be assigned to the numeric variable indicated until a comma or an EOL character is

encountered. Numeric values can be ended either with the EOL character or the comma. Commas can not be used to end string values, as they are regarded as part of the string.

If no data is available to be read into the numeric variable, or if the data is invalid, an error will result.

When a string *variable* is specified, the data being input will be interpreted as string data. If no characters are read, the string variable will be assigned the null value. If more characters are read than allowed for in the string variable's DIM statement, the INPUT# statement will disregard the excess characters. The EOL character will end the string input.

The GET statement is used to read a single numeric value via the open channel specified. GET uses the following configuration.

GET #*channel*, *numeric variable*

The *channel* specified must be open for input. The *numeric variable* indicated will accept the value returned by GET. This value will lie between 0 and 255.

When GET is used with the 410 Program Recorder and the buffer is empty, the initial GET statement will result in a block of data being read into the cassette buffer. The first value in the buffer will be assigned to the *numeric variable* specified with the first GET statement.

Each successive GET statement will read a value from the cassette buffer. When the buffer has been emptied, another block of data will be read into the cassette buffer from the cassette tape.

CHAPTER 7. ATARI 810 DISK DRIVE

Introduction

The Atari 810 Disk Drive is used for storing BASIC programs or data files on floppy diskettes.

A disk stores data in a magnetic form, much like data is stored on magnetic tape. The main difference between storage on a magnetic tape and storage on a disk is that the disk surface is round--much like a record's surface.

The disk drive contains a device known as a **read/write head**, which is used to read and write information. The computer can move the head to any position desired on the disk surface. This is in contrast to magnetic tape, where data is read from or written onto the tape in consecutive order.

This capacity to read or write data at a particular position is known as **random access**. Disk drives are known as random access storage devices. On the other hand, in cases where data must be read or written in a consecutive order, the accessing is known as **sequential access**. A cassette tape recorder is known as a sequential access drive.

Types of Disks

There are three primary types of disks used by microcomputers; **hard disk**, **Winchester disks**, and **floppy diskettes**. These will be described in the following sections.

Hard Disks

Microcomputer hard disk systems generally allow storage of 5 to 30 megabytes of data. One megabyte is the equivalent of one

million bytes. The hard disk itself is made of a rigid material with a magnetic coating. The disk drive and the hard disk are separate units. The operator can remove one hard disk and replace it with another.

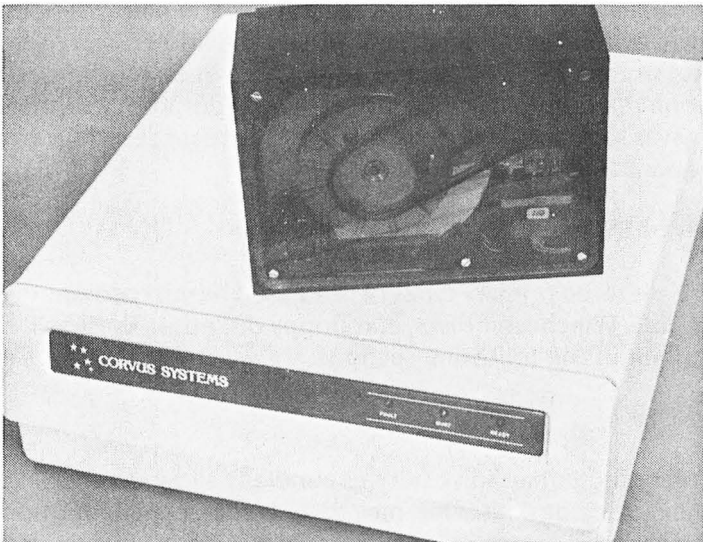
Winchester Disk Drives

Winchester disk drives are designed so that from 6 to 10 times more data can be stored on their surface than on a standard floppy diskette. Winchester disks must be kept very clean as they are extremely vulnerable to dust, dirt, and smoke.

Since they must be kept so clean, Winchester disks must be sealed inside of the disk drive. This means that Winchester disks cannot be changed.

Since Winchester disks cannot be removed, floppy disk systems often are used in conjunction with Winchester disks to allow for back-up storage. Winchester disk systems are generally used with microcomputers rather than hard disk systems. A Winchester drive is shown in Illustration 7-1.

Illustration 7-1. Winchester Disk System



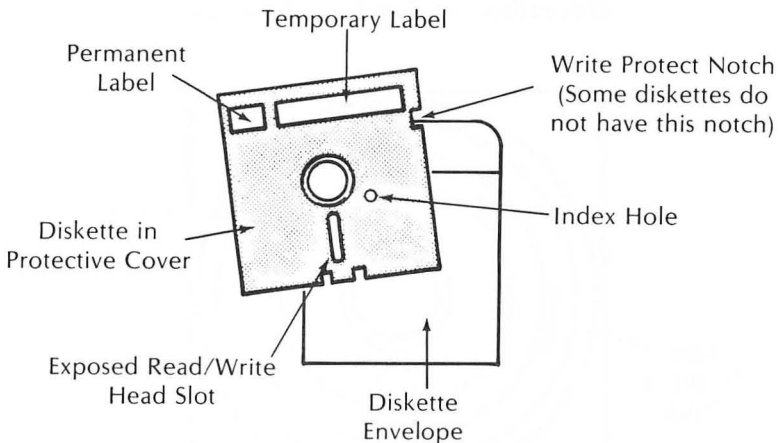
Floppy Diskettes

The most widely used type of disk storage with microcomputers is floppy disk storage. A floppy diskette consists of a round vinyl disk which is enclosed within a plastic cover. The diskette is generally stored in a diskette envelope.

This cover protects the diskette from damage while it is being handled by the operator. The diskette should never be removed from its cover. A 5¼ inch diskette with its protective envelope is shown in Illustration 7-2.

The diskette is allowed to rotate within the protective envelope. The round hole in the middle of the diskette allows the disk drive to hold the diskette and spin it. The oblong shaped opening on the protective envelope provides an area where the head can read from or write to the diskette surface.

Illustration 7-2. Mini-Floppy Diskette



Floppy diskettes come in two sizes: 8 inch and 5¼ inch. The 5¼ inch diskettes are also known as mini-floppy diskettes. The Atari 810 Disk Drive uses mini-floppy diskettes.

Tracks & Sectors

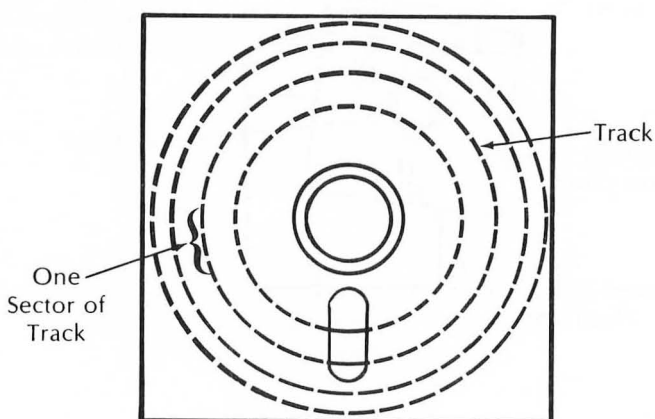
To facilitate the process of searching for data on the diskette surface, that surface is divided into tracks and sectors.

Tracks may be visualized as a series of concentric circles on the diskette surface, as shown in Illustration 7-3. Atari's DOS divides a diskette into 40 tracks.

To further reduce the time necessary to search for a particular data item, Atari's DOS divides each track into 18 sectors, which are also shown in Illustration 7-3.

With 40 tracks available and 18 sectors per track, Atari DOS divides each diskette into 720 sectors. However, 18 of these 720 sectors are used by Atari DOS, and cannot be used to store programs or data.

Illustration 7-3. Tracks and Sectors



Each individual sector holds 128 bytes of data. When DOS has access to the track and sector where a particular data item is being stored, it will only have to search 128 bytes to find that item. The result of dividing the diskette surface into tracks and sectors is that access time is greatly decreased.

Hard and Soft Sectors

Locating a particular track on the disk surface is a relatively uncomplicated matter. The drive merely moves the head to the position on the diskette where the specified track is located, much like the needle on a phonograph is positioned to the location of a specific song on a record album.

However, locating a particular sector is a more difficult process. Two different methods are used to locate sectors on a disk; hard sectoring and soft sectoring.

Both the hard and soft sector methods involve the use of an index hole. The index hole is shown in Illustration 7-2. It is located just to the right of the large hole in the middle of the 5¼ inch diskette.

The index hole, as shown in Illustration 7-2, is a hole only in the diskette's protective covering. Another index hole is located on the actual diskette surface inside the envelope. As the diskette spins, the index hole (or holes) on the diskette surface passes underneath the hole in the protective envelope.

A light source inside the disk drive shines light onto the area of the diskette containing the index hole. When an index hole on the disk surface is aligned with the index hole on the protective envelope, the light will shine through to a sensor. The sensor will relay information on the location of the index holes, which can be used to calculate the various sector locations.

Now that we have discussed the concepts of locating sectors, we will discuss the difference between hard and soft sectored diskettes. A hard sectored diskette contains a number of holes, each of which indicates the location of a sector. An extra hole is used to indicate the location of the first sector. The location of

the various sectors is determined by counting the number of holes occuring after the first sector. A hard sectored diskette is depicted in Illustration 7-4.

Soft sectored diskettes have only one index hole as shown in Illustration 7-5. This solitary index hole marks the location of the first sector. By timing the rotation speed of the floppy diskette, the location of the other sectors can be determined. The Atari 810 uses soft-sectored diskettes.

Illustration 7-4. Hard Sectored Diskette

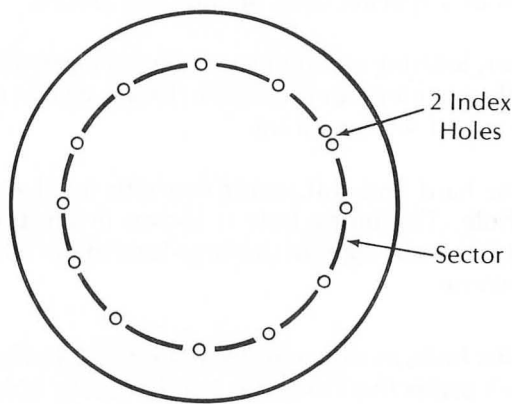
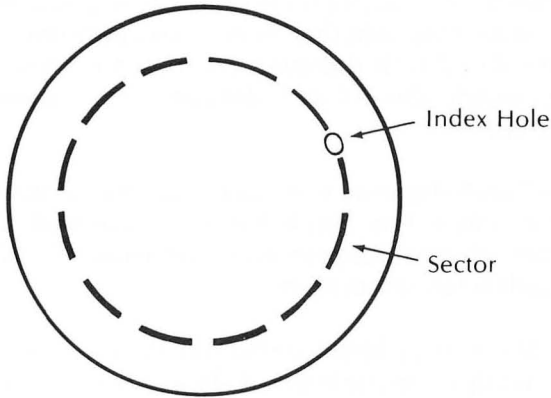


Illustration 7-5. Soft Sectored Diskette



Single and Double Sided Diskettes

Some floppy diskettes are designed to be written on only one side. These are known as single sided (SS) diskettes.

Diskettes which are designed to be written on both sides are known as double sided (DS) diskettes.

Single, Double and Quad Density Diskettes

Density refers to a diskette's recording format, which in turn affects its capacity. Single density 5¼ inch diskettes have roughly 94K of capacity, double density 5¼ inch diskettes have a capacity of about 140-160K, and quad density 5¼ inch diskettes have a capacity of up to 370K.

The Atari 810 uses single sided single density diskettes. The Atari 810 has a total storage capacity of 88,375 bytes. This figure is calculated by multiplying the 707 sectors available for data storage by 125 bytes per sector. Three bytes per sector are allocated for the File Management Subsystem.

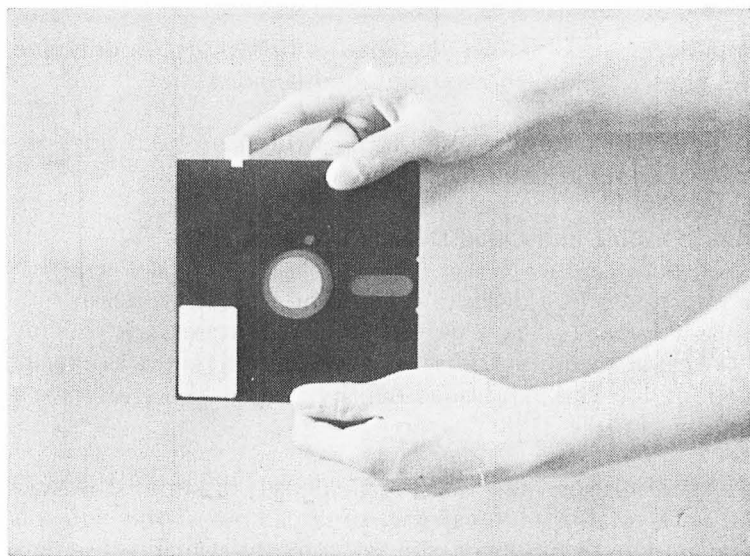
Diskette Write Protection

Diskettes have a notch on the side of their protective envelope which determines whether or not data can be written onto that diskette. On 8 inch diskettes, this notch is known as a write-protect notch. On 5¼ inch diskettes, it is known as a write-enable notch.

On an 8 inch diskette, information cannot be written onto the diskette unless this notch has been covered. On 5¼ inch diskettes, information cannot be written onto the diskette unless the notch is left uncovered.

Some 5¼ inch diskettes (especially system diskettes) may be permanently write protected if their protective envelope does not contain a notch. Any 5¼ inch diskette with a notch can be write protected by merely covering the notch with a piece of tape as shown in Illustration 7-6.

Illustration 7-6. Write Protecting a 5¼ Inch Diskette



Disk Files

The Atari 810 stores data in **files**. A disk file can contain either a BASIC program, a machine language program, or data. An Atari diskette can contain as many as 64 files.

Files are assigned unique **filenames** of up to eight characters. The first character of a filename must be a capital letter. Subsequent filename characters can either be capital letters or numbers. Blank spaces, punctuation marks, and special characters (#, \$, @) are not allowed in filenames.

Filenames can also contain a **filename extension** of three or fewer characters. The filename and filename extension must be separated with a period. Filename extensions can contain either letters or numbers.

Filename extensions are often used to indicate the type of file. The implied meanings of commonly used filename extensions are listed in Table 7-1.

Filename Match Characters

Atari DOS allows the use of the filename match characters, ? and *. These characters can be used to stand for any single character (?) or group of characters (*). For example, FILE?.DAT would match the following filenames.

FILE1.DAT
FILEZ.DAT

FILE?.DAT would not match the following filenames:

FILE.DAT
FILE1.BAS

FILE.* would match any of the following filenames.

FILE.BAS
FILE.TXT
FILE.DAT

Filename match characters are not allowed in Atari DOS version 1.0, and can only be used with the following DOS menu options in DOS version 2.0.

- A. Disk Directory
- B. Copy File
- D. Delete File
- E. Rename File
- F. Lock File
- G. Unlock File
- O. Duplicate File

Table 7-1. Filename Extensions & Type of File

Filename Extension	File Type
ASM	Assembly language source file.
BAK	Backup File.
BAS	File containing a Basic program in tokenized format.
DAT	Data File.
OBJ	Assembly language program assembled into machine language. Also known as an object file.
TXT	Text File.
SYS	System file. Used with system programs such as DOS programs or the BASIC language interpreter.

ATARI DOS (Disk Operating System)

An operating system can be defined as a group of programs which manage the computer's operation. A disk operating system can be defined as a group of programs that manage the transfer of data to and from a storage device such as disk or magnetic tape.

Two different versions of Atari DOS currently exist, DOS 1.0 and DOS 2.0. The version your system uses should be marked on your system diskette.

The major differences between DOS 1.0 and 2.0 are listed in Table 7-2.

Table 7-2. DOS Version 1.0 & 2.0 Differences

DOS 1.0	DOS 2.0
11 second boot time	7 second boot time
No filename match characters	Filename match characters allowed during certain DOS operations
MEM.SAV not available	MEM.SAV allows additional memory space
No AUTORUN.SYS	AUTORUN.SYS allows a file to be loaded and executed upon boot
No appending of files	SAVE BINARY FILE "/A" option allows appending of two files
Bad diskette sectors not indicated during formatting	A diskette with bad sectors cannot be formatted

Menu must be displayed before a new command can be entered	Menu need not be re-displayed to enter a new command
DOS files can only be written to Drive 1	DOS files can be written to any drive
NOTE/POINT not available for random access	NOTE/POINT statements available for random access
A maximum of 3 files can be open at any one time	Up to 8 files can be open at any one time

Two Parts of Atari DOS

Atari DOS consists of two different parts. One part is used to save and load BASIC programs and to read and write data files. The other part of Atari DOS consists of a group of utility programs used in performing operations with disk files as well as reading and writing machine language files.

In DOS version 1.0, both parts of DOS are stored as a single file on disk. This file is named DOS.SYS.

In DOS version 2.0, the two parts of DOS are stored on disk as two separate program files. The first part (DOS.SYS) is needed whenever the disk drive is being used. The second part (DUP.SYS) is only required when the disk utilities are being used.

By separating DOS in version 2.0, better use can be made of the Atari's memory. Only the part of DOS required need be loaded into memory. The portion of memory normally used by the other part of DOS is freed for use.

Disk Buffer

Atari DOS controls the transfer of information between the Atari computer and the disk drive. Information is transferred in 128 character blocks.

Four separate portions of Atari RAM are set aside to hold data being transferred to a disk from a disk drive. These are known as **disk** buffers. The reason why there are four disk buffers is that as many as four disk drives can be attached to the Atari at any one time.

When Atari DOS is instructed to supply data, it will first attempt to supply this data from the disk buffer. When the buffer runs out of data, another block of data will be read into the buffer from the diskette.

Data is also written to diskette from the disk buffer one block at a time. Information that is to be transferred to disk is first sent to the disk buffer. When the disk buffer has been filled, Atari DOS writes the data in the buffer to the diskette.

Booting DOS

Before Atari DOS is available for usage, it must be loaded into memory from a diskette. This process is known as **booting** DOS. The procedure for booting DOS is as follows:

1. Power on Drive 1. If your system includes more than one drive, Drive 1 can be determined by examining the access hole on the back of each drive. On Drive 1, both the black and white switches are located to the far left.
2. Insert a diskette containing a copy of DOS into Drive 1.
3. Turn the Atari 400 or 800's power off (if necessary) and on.
4. If you are booting DOS version 1.0 or if you are booting DOS version 2.0 on a system without a BASIC ROM cartridge installed, the DOS Menu will appear on the video display.

5. If you are booting DOS version 2.0 on a system with a BASIC ROM cartridge installed, the BASIC prompt READY will appear on the video display. By entering the following command at the keyboard,

DOS 

the DOS Menu will be loaded.

In step 5, DOS.SYS is loaded during the initial part of the loading process. When the DOS statement is executed, DUP.SYS will be loaded into memory.

In DOS version 2.0, DUP.SYS is loaded into an area of memory where BASIC programs are stored. When DUP.SYS is loaded, any existing BASIC program in memory will be erased.

In DOS version 1.0 the disk utility package is contained in the DOS.SYS file. DOS.SYS does not overwrite the area in RAM where BASIC programs are stored. Therefore, loading DOS version 1.0 will not affect a BASIC program in memory.

DOS Menu

The DOS Menu loaded depends on the version of DOS used. The DOS Menu loaded by DOS version 1.0 is depicted in Illustration 7-7. The DOS Menu loaded by DOS version 2.0 is pictured in Illustration 7-8.

Illustration 7-7. DOS Version 1.0 Menu

DISK OPERATING SYSTEM 9/24/79
COPYRIGHT 1979 ATARI

A. DISK DIRECTORY	I. FORMAT DISK
B. RUN CARTRIDGE	J. DUPLICATE DISK
C. COPY FILE	K. BINARY SAVE
D. DELETE FILE(S)	L. BINARY LOAD
E. RENAME FILE	M. RUN AT ADDRESS
F. LOCK FILE	N. DEFINE DEVICE*
G. UNLOCK FILE	O. DUPLICATE FILE
H. WRITE DOS FILE	
SELECT ITEM	

■

*N--DEFINE DEVICE is not used.

Illustration 7-8. DOS Version 2.0 Menu

DISK OPERATING SYSTEM II VERSION 2.0S
COPYRIGHT 1980 ATARI

A. DISK DIRECTORY	I. FORMAT DISK
B. RUN CARTRIDGE	J. DUPLICATE DISK
C. COPY FILE	K. BINARY LOAD
D. DELETE FILE(S)	L. BINARY LOAD
E. RENAME FILE	M. RUN AT ADDRESS
F. LOCK FILE	N. CREATE MEM.SAV
G. UNLOCK FILE	O. DUPLICATE FILE
H. WRITE DOS FILES	
SELECT ITEM OR [RETURN] FOR MENU	

■

As shown in Illustrations 7-7 and 7-8, a number of different disk operations are available on the DOS Menu. To choose an operation from the DOS Menu, enter the letter corresponding to that operation and press Return.

Once the disk operation has been chosen, a prompt message for that operation will appear on the screen. Generally, this prompt message specifies some additional information required by the disk operation.

Once the disk operation has been chosen, the prompt,

SELECT ITEM (DOS 1.0)

or

SELECT ITEM OR [RETURN] FOR MENU

will once again appear on the bottom of the video display.

If another item is specified, that disk operation's prompt will be displayed. If Return is pressed, the DOS Menu will be redisplayed.

DOS MENU OPERATIONS

In the following sections, we will discuss the various DOS Menu Operations.

A. Disk Directory

The Disk Directory operation lists the files present on a diskette.

When the Disk Directory operation has been specified by entering A and pressing Return, the following prompt will appear on the video display:

DIRECTORY—SEARCH SPEC, LIST FILE

If the Return key is pressed in response to this prompt, the names

of each file on the diskette in drive 1 will be displayed on the screen followed by the size of the file (in sectors). The last line of the directory listing will contain the number of unused sectors on the diskette. A sample directory listing is pictured in Illustration 7-9.

As previously mentioned, pressing Return in response to the SEARCH SPEC, LIST FILE prompt will cause all files on the diskette in drive 1 to be listed. When Return is pressed in response to this prompt, DOS will assume the default values for the SEARCH SPEC and LIST FILE parameters.

SEARCH SPEC indicates the file specification of any specific files to be listed by the Directory operation. This file specification consists of the capital letter D followed by the number of the disk drive whose diskette is to be searched, followed by the name of the file or files to be searched for. The drive identifier and filename should be separated by a semicolon.

Illustration 7-9. Directory Listing



*DOS	SYS	039
*DUP	SYS	042
DISP	OBJ	001
PROGRAM2	BAS	012
PROGRAM3	BAS	013
600 FREE SECTORS		

If the drive number is omitted, DOS will assume drive 1 is to be searched. In other words D1 is the **default** value for the drive identifier.

Filename match characters can be used in the filename portion of the SEARCH SPEC parameter. For instance, the following entry,

D2:*.DAT

would cause all files on drive 2 with the filename extension DAT to be listed. The default value for the filename portion of the SEARCH SPEC parameter is *.*. This value causes all files to be listed, as *.* matches all filenames.

The second disk directory parameter, LIST FILE, specifies the device where the directory output is to be listed. The default value for the output device is E:, which indicates the video screen.

If you wish to have the directory listing sent to the printer, use P: as the LIST FILE parameter. For example, the following entry,

D2:*.DAT,P:

would cause all files on drive 2 with the extension DAT to be listed by the printer.

When using the LIST FILE option, be certain to separate your entry from the SEARCH SPEC entry with a comma.

B. Run Cartridge

When the Run Cartridge operation is chosen from the DOS Menu, DOS will return control of the Atari computer to the cartridge inserted in the unit. If the BASIC cartridge is inserted, the BASIC prompt,

READY

will be displayed on the screen. If the Assembler Editor cartridge

is inserted, the prompt,

EDIT

will be displayed.

If a cartridge is not inserted in the Atari, the following message,

NO CARTRIDGE

will appear on the screen when the Run Cartridge operation is chosen. Another operation must then be chosen from the DOS Menu.

If you are using DOS version 2.0, the Run Cartridge operation should not be used to return to BASIC when the MEM.SAV file exists on the diskette. Instead, the System Reset key should be pressed to return to BASIC. By following this procedure, data will be correctly returned into memory from MEM.SAV. MEM.SAV will be discussed in more detail later in this chapter.

C. Copy File

The Copy File disk operation is used on Atari systems with two or more disk drives to copy a file from the diskette in one drive to a diskette in another drive. Copy File can also be used to create a back-up copy of a file on the same diskette with a different filename.

When Copy File is executed, the following prompt will appear on the video display:

COPY-FROM, TO?

The FROM parameter specifies the file or files to be copied. The FROM parameter generally consists of a file specification, but can also be a device name such as the video screen (E:).

Filename match characters can be used in the file specification used for the FROM parameter.

The TO parameter specifies the destination of the file or files being copied. Again, the TO parameter generally consists of a file specification, but can also be a device such as the printer (P:), screen (E:), or disk drive (D:).

The Copy File operation cannot be used to copy the DOS.SYS file. Any attempt to do so will result in an error message. The DOS.SYS file can be copied using the Write DOS.SYS operation (H.).

If the source file specified does not exist, error 170 (FILE NOT FOUND) will appear on the screen. If the destination diskette's directory already contains 64 filenames, error 169 (DIRECTORY FULL) will appear. If there are not enough free sectors on the destination diskette for the copy operation to take place, error 162 (DISK FULL) will appear.

If you are using the Copy File operation in DOS version 2.0 and a MEM.SAV file exists on the system diskette, a second prompt message (as shown below) will appear before the Copy File operation is executed.*

TYPE "Y" IF OK TO USE PROGRAM AREA
CAUTION: A "Y" INVALIDATES MEM.SAV

If the user's response to the preceding prompt is Y, Copy File will use the entire user program area for the copying process which invalidates the MEM.SAV file.

A response of N instructs DOS to use a smaller internal buffer for the Copy Files operation. The MEM.SAV file will be retained. However, the copying process will be slower.

The Copy File operation can be used with the Append option (/A) to add one file to the end of another file. This process is known as **merging**.

For example, the following parameter entry would cause FILEA.TXT to be merged with FILEB.TXT.

*This prompt may not appear in some systems.



COPY--FROM, TO?
D1:FILEA.TXT, D2:FILEB.TXT/A

The Append option should not be used with BASIC program files stored in tokenized format.

Illustration 7-10 contains examples of the usage of the Copy File operation.

Illustration 7-10. Copy Files Example



Example 1

SELECT ITEM OR [RETURN] FOR MENU
C 
 COPY--FROM, TO?
D1:FILEA.TXT, D2:FILEA.TXT 

 SELECT ITEM OR [RETURN] FOR MENU

In the preceding example, FILEA.TXT is copied from the diskette in drive 1 to the diskette in drive 2, using the same filename.

Example 2

SELECT ITEM OR [RETURN] FOR MENU
C 
 COPY--FROM, TO?
D1:FILEA.TXT, D1:FILEB.TXT 

 SELECT ITEM OR [RETURN] FOR MENU

In Example 2, a copy of FILEA.TXT is created on the diskette in drive 1 and is assigned a new filename FILEB.TXT.

Example 3

SELECT ITEM OR [RETURN] FOR MENU

C 

COPY-FROM, TO?

D1:PROGA.BAS,P: 

SELECT ITEM OR [RETURN] FOR MENU

In Example 3, PROGA.BAS is listed on the printer.

Example 4

SELECT ITEM OR [RETURN] FOR MENU

C 

COPY-FROM, TO?

E:,D1:TEXTC.DAT 

PEAR 

APPLE 

BANANA 

GRAPES 

Control-3

SELECT ITEM OR [RETURN] FOR MENU

In Example 4, the data displayed on the screen will be copied into TEXTC.DAT on drive 1. When Control-3 is pressed, the entry will be ended.

D. Delete File

The Delete File operation allows the user to delete unneeded files from the diskette and the disk directory.

When the Delete File operation is chosen, the following prompt will appear on the video display.

DELETE FILESPEC

The file specification should be entered. Filename match characters may be used in the file specification.

Once the file specification has been entered, a second prompt will be displayed.

TYPE "Y" TO DELETE...
FILENAME

FILENAME will be replaced with the filename of the file to be deleted. If the user enters Y followed by Return, the file will be deleted. If N or any other letter is entered followed by Return, the file will not be deleted.

If the file specification entered in response to the DELETE FILESPEC prompt matches more than one filename on the diskette, each matching filename will be displayed. The user must enter Y following each filename for the deletion to occur.

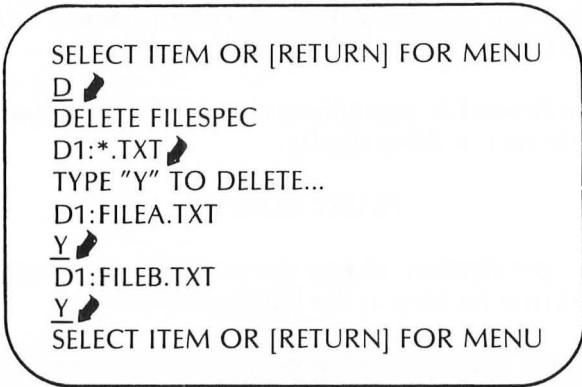
If the No verification option (/N) is specified in response to the DELETE FILESPEC prompt, the second prompt will not appear. The files specified will automatically be deleted.

A file which has been locked cannot be deleted using the Delete File operation. Any attempt to do so will result in error 167.

Examples of the use of the Delete File operation are given in Illustration 7-11.

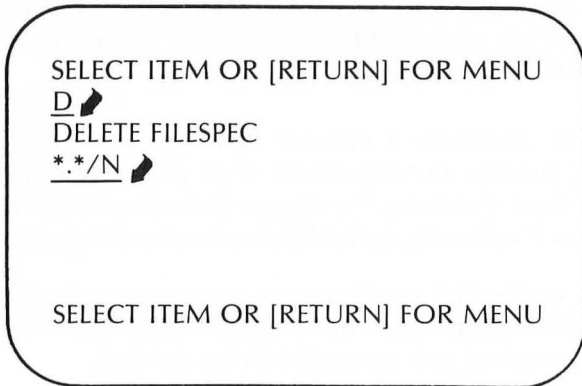
Illustration 7-11. Delete File Example

Example 1



In the preceding example, any files with an extension of .TXT will be prompted for deletion. FILEA.TXT and FILEB.TXT will both be deleted from the diskette in drive 1.

Example 2



In Example 2, all files on the diskette in drive 1 will be deleted. Note the use of the No Verification option to suppress the second prompt.

E. Rename File

The Rename File operation can be used to change the name of any file on the diskette. Be careful not to use Rename File to

change the name of DOS.SYS. If DOS.SYS is renamed, the DOS menu will no longer load.

When Rename File is specified, the following prompt will appear.

RENAME, GIVE OLD NAME, NEW

OLD NAME will consist of the file specification of the file to be renamed. If a drive identifier is not included in the file specification, drive 1 will be assumed.

The NEW NAME will consist of the new filename for the file specified in OLD NAME. Filename match characters can be used with both the OLD NAME and NEW NAME parameters.

A locked file cannot be renamed. Any attempt to do so will result in error 167 (File Locked).



A file on a diskette that has been write-protected cannot be renamed. Any attempt to do so will result in error 144 (Device Done Error).

Also, if the user attempts to rename a file that does not exist on the diskette, error 170 will occur (File Not Found).

Examples of the use of the Rename File operation are given in Illustration 7-12.

Illustration 7-12. Rename File Examples



Example 1

SELECT ITEM OR [RETURN] FOR MENU
E 
RENAME-GIVE OLD NAME, NEW
TEXTA.DAT, TEXTB.DAT 

SELECT ITEM OR [RETURN] FOR MENU

In Example 1, TEXTA.DAT on drive 1 is renamed to TEXTB.DAT.

Example 2

SELECT ITEM OR [RETURN] FOR MENU
E 
RENAME--GIVE OLD NAME, NEW
D2:*.BAS,*.BAK 

SELECT ITEM OR [RETURN] FOR MENU

In Example 2, all files on drive 2 with the extension .BAS will be renamed with the extension .BAK.

F. Lock File

The Lock File operation write protects a file. When a file is locked, it cannot be written to, renamed, deleted, or appended to. If an attempt is made to do so, error 167 (File Locked) will appear.

Locked files appear in the Disk Directory with an asterisk before the filename.

When the Lock File operation is specified, the following prompt will appear.

WHAT FILE TO LOCK?

Enter the file specification of the file to be locked. Filename match characters may be used to lock multiple files with a single file specification.

It is good practice to lock the DOS.SYS and DUP.SYS files.

G. Unlock File

The Unlock File operation is used to unlock one or more disk files previously locked with Lock File. When the Unlock File operation is specified, the following prompt will appear:

WHAT FILE TO UNLOCK?

The file specification of the file to be unlocked should be entered. Filename match characters can be used to specify more than one file.

H. Write DOS File

The Write DOS File operation allows the user to copy DOS 1.0 or 2.0 onto a diskette. In DOS version 1.0, the DOS.SYS file is copied. In DOS version 2.0, the DOS.SYS and DUP.SYS files are copied.

DOS is copied from the computer's memory, not directly from a diskette during the Write DOS File operation.

In DOS 1.0, the following prompt appears when Write DOS File is specified.

TYPE "Y" TO WRITE NEW DOS FILE

DOS.SYS will be written to the diskette in drive 1.

In DOS 2.0, the following prompt will appear.

DRIVE TO WRITE FILES TO?

Here, the operator should enter the drive number where DOS should be copied. This can be either drive 1, 2, 3, or 4. Once the drive number has been entered, the following prompt will appear.

TYPE "Y" TO WRITE NEW DOS FILE

If a Y is entered, DOS will be written to the diskette in the specified drive. Any other entry aborts the Write DOS File operation.

I. Format Diskette

All blank diskettes must be **formatted** before they can be used by DOS. Formatting is a process where a pattern is recorded on the diskette which allows data to be written to or read from its surface. The Atari 810 Disk Drive requires approximately two minutes to format a diskette.

When the Format Diskette operation is specified, the following prompt will appear.

WHICH DRIVE TO FORMAT?

The user should specify the number of the drive containing the diskette to be formatted. A second prompt will then appear.

TYPE "Y" TO FORMAT DRIVE 1*

If the user responds to this prompt with a Y, the diskette in the drive specified will be formatted. Any other entry will abort the Format Diskette operation.

*assuming drive 1 was specified in the first prompt.

If a diskette contains bad sectors, DOS will not format it. After the initial discovery that the diskette contains bad sectors, DOS will attempt to format the diskette two more times. Upon the third unsuccessful attempt, error 173 (Bad Sectors at Format Time) will be displayed.

Be certain that you do not format a diskette that contains data you wish to retain. Formatting a diskette destroys any existing data on that diskette.

J. Duplicate Disk

The Duplicate Disk operation allows an entire diskette to be copied. This operation can be used with Atari systems with either one or two disk drives.

When Duplicate Disk is specified, the following prompt will appear.

DUP DISK--SOURCE, DEST DRIVES?

The user should respond with the drive number containing the diskette to be copied, and the drive number which will contain the diskette in which the copy is to be made.

If your Atari system has only one drive, you should respond to this prompt with an entry of 1,1.

The following prompt will then be displayed.

INSERT SOURCE DISK, TYPE RETURN

The user should then insert the diskette to be copied in the sole disk drive and press Return. A portion of the data stored on the diskette will then be read into the Atari's memory. The following prompt will then be displayed.

INSERT DESTINATION DISK, TYPE RETURN

The user should then replace the diskette being copied with a blank formatted diskette and press Return.

The data held in the Atari's RAM will be written to the destination diskette, after which the Insert Source Disk prompt will reappear. Continue this process until the entire diskette has been copied.

If your Atari system contains multiple drives, the duplication process is much more simple. When different source and destination drives are specified (ex. 1,2), the following prompt will be displayed.

INSERT BOTH DISKETTES, TYPE RETURN

After inserting the diskette to be copied in the source drive and the blank diskette on which the copy is to be made in the destination drive, press Return and the duplication process will begin. The duplication process can take several minutes if the source file is filled.

It is a good practice to write protect the source diskette to prevent it from being accidentally overwritten if an error is made.

With DOS version 2.0 systems, the following prompt is displayed.*

TYPE "Y" IF OK TO USE PROGRAM AREA?
CAUTION: A "Y" INVALIDATES MEM.SAV

If Y is entered, the user program area will be used for the copying process, and existing programs in memory will be erased. An entry other than Y causes Duplicate Disk to be aborted. If a program is stored in RAM that you wish to save, it should be copied to cassette or diskette before the Duplicate Disk operation is begun.

K. Binary Save

The Binary Save operation is used to save the contents of RAM on disk in object file format. This format is also used for programs written using the Assembler Editor cartridge.

When the Binary Save operation is specified in DOS 2.0, the

*This prompt may not appear in some systems.

following prompt will be displayed.

SAVE--GIVE FILE, START, END (,INIT, RUN)

FILE is the name of the file to be saved. A drive specifier may be included.

The START and END parameters are required for either a binary file or a program. These specify the starting and ending addresses in hexadecimal of the portion of the memory to be saved.

The INIT and RUN addresses are optional parameters. These allow a program to be executed upon loading. The INIT address gives the starting address of an initialization routine. The RUN address gives the starting location of the main program. The INIT and RUN addresses are used by the Binary Load operation to automatically execute a program after it has been loaded. The INIT and RUN addresses must be specified in hexadecimal notation.

When the Binary Save operation is specified in DOS 1.0, the following prompt will appear.

SAVE--GIVE FILE, START, END

Again, FILE gives the name of the object file to be created. A drive specifier can be used preceding the filename.

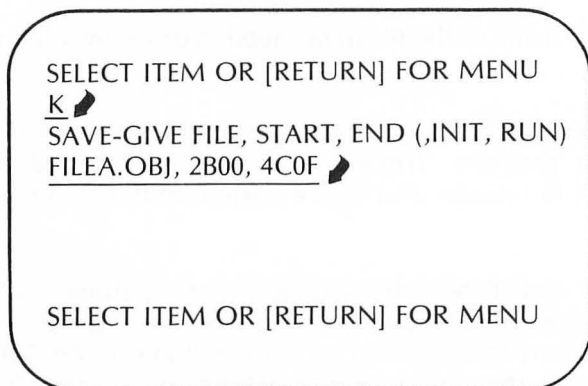
START and END give the beginning and ending addresses in hexadecimal of the block of data to be saved.

A file can be saved in DOS 1.0 so that it will be automatically executed when it is loaded by Binary Load. To accomplish this, the program starting address should be placed in memory addresses 736 and 737 (2E0 and 2E1 in hexadecimal). The low byte of the program starting address should be placed in address 736 and the high byte in 737. The POKE statement can be used to place the proper values in these locations.

Illustration 7-13 gives an example of the use of the Binary Save

operation.

Illustration 7-13. Binary Save Example



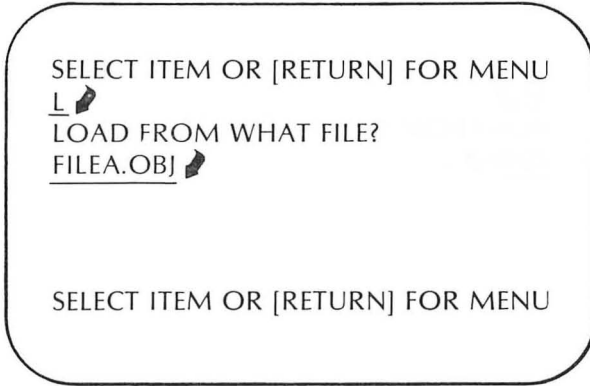
In the preceding example, the contents of memory locations beginning at 2B00 and ending at 4C0F will be saved in a file named FILEA.OBJ on drive 1.

L. Binary Load

The Binary Load operation is used to load a file created with Binary Save or an assembly language object file into RAM. If the RUN and INIT addresses were appended to the file, the file will execute upon loading.

If the /N option is specified, the INIT and RUN addresses will be disregarded, and the file must be run using the DOS Menu's Run At Address operation. Also, files without an INIT or RUN address must be run using the Run At Address operation.

An example of a Binary Load operation is given in Illustration 7-14.

Illustration 7-14. Binary Load Example

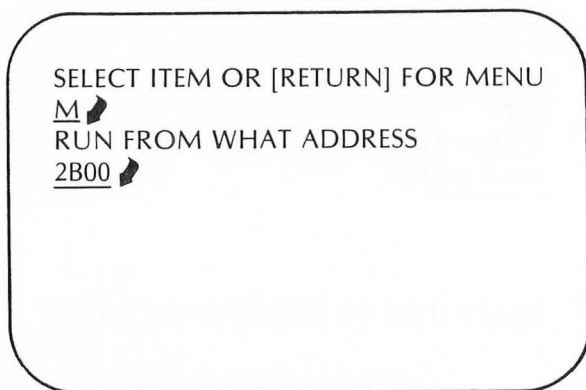
In some situations in DOS 2.0, the binary file may require a portion of the memory area used by DUP.SYS. If this occurs, the portion of the binary file that requires memory used by DUP.SYS will be saved on the MEM.SAV file until the binary file has been executed. If the MEM.SAV file is not present, the following message will appear.

NEED MEM.SAV TO LOAD FILE

M. Run At Address

The Run At Address operation is used to execute a machine language program in memory by entering its hexadecimal starting address. An example of the use of the Run At Address operation is given in Illustration 7-15.

Illustration 7-15. Run At Address Example



N. Create MEM.SAV*

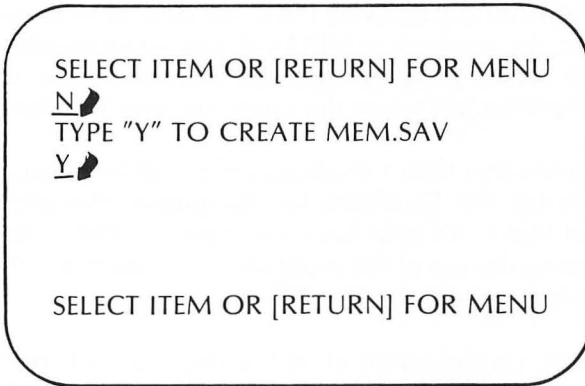
The Create MEM.SAV operation is used to create a MEM.SAV file on the diskette in drive 1.

Whenever DOS version 2.0 is booted, the DUP.SYS disk file will overwrite an area in memory where BASIC programs are stored. When a MEM.SAV file is present on the diskette in drive 1, the computer will transfer all data present in the memory area used by DUP.SYS into the MEM.SAV file. Afterwards, DUP.SYS will be loaded, and the DOS menu will appear.

When you have finished using DOS, by entering the Run Cartridge operation, the program in MEM.SAV will be automatically loaded from MEM.SAV into RAM.

When a MEM.SAV file is used, more time will be required to load the DOS menu. Illustration 7-16 depicts the use of Create MEM.SAV.

*Define Device (Item N. in DOS version 1.0) is not utilized.

Illustration 7-17. Create MEM.SAV Example

If the user attempts to create a MEM.SAV file on a diskette which already contains a MEM.SAV file, the following will be displayed on the video screen.

MEM.SAV FILE ALREADY EXISTS
 SELECT ITEM OR RETURN FOR MENU

O. Duplicate File

The Duplicate File operation is used to copy file from one diskette to another in systems with only one drive.

When the Duplicate File operation is specified, the following prompt will appear.

NAME OF FILE TO MOVE?

Since the source and destination files will be the same, only one filename need be entered. Also, since the system includes only one disk drive, a drive identifier is not necessary. Filename match characters may be used in the filename entry.

In DOS version 2.0, the following prompt will then appear.*

TYPE "Y" IF OK TO USE PROGRAM AREA
 CAUTION: A "Y" INVALIDATES MEM.SAV

*This prompt may not appear in some systems.

If a Y is entered, the entire program area of memory will be used for the file duplication process. This will speed the duplication process. However, by allowing the program area to be used for duplication, the contents of MEM.SAV cannot be rewritten into RAM. Any BASIC program that you intended to save using MEM.SAV will be lost when the system returns to BASIC.

Any response other than Y disallows the use of the program area of memory for the Duplicate File operation. This allows the contents of MEM.SAV to be later rewritten into RAM. However, by disallowing the use of the program area of memory, the time necessary to duplicate the file will increase.

Be extremely careful when using the Duplicate File operation with DOS 1.0. When this operation is specified in DOS 1.0, the program area of memory will be erased. Any existing files will be lost after the duplication process has been completed.

An example of the use of the Duplicate File option is given in Illustration 7-18. Notice that the diskettes may have to be swapped several times in order to complete the duplication process.

USING BASIC PROGRAMS ON DISKETTE

In the following sections, we will discuss the BASIC statements used to save programs on diskette and then reload these programs back into memory.

Saving BASIC Programs On Diskette

Once a BASIC program has been entered into RAM via the keyboard, it must be stored on diskette or it will be lost when the Atari's power is turned off or when a NEW statement is executed. The SAVE and LIST statements are used to save a BASIC program on diskette.

The SAVE statement uses the following configuration when used with the Atari 810 Disk Drive.

Illustration 7-18. Duplicate File Example

```

SELECT ITEM OR [RETURN] FOR MENU
O
NAME OF FILE TO MOVE?
TEXT?.DAT
TYPE "Y" IF OK TO USE PROGRAM AREA
CAUTION: A "Y" INVALIDATES MEM.SAV
Y
INSERT SOURCE DISK, TYPE RETURN
COPYING--D1:TEXTA.DAT*
INSERT DESTINATION DISK, TYPE RETURN
INSERT SOURCE DISK, TYPE RETURN
COPYING--D1:TEXTB.DAT*
INSERT DESTINATION DISK, TYPE RETURN
INSERT SOURCE DISK, TYPE RETURN
SELECT ITEM OR [RETURN] FOR MENU

```

*This message appears only if more than one file is copied.

SAVE "D#:filename"

where *D* is the device name for the Atari 810 disk drive. This parameter is required. # indicates the drive number (1, 2, 3, or 4). If # is omitted, drive 1 is assumed. The BASIC program *filename* is separated by a colon from the device name and drive number.

The SAVE statement stores BASIC programs in tokenized format, keywords are abbreviated as one character tokens.

The LIST statement uses the following configuration,

```
LIST "D#:filename"[,beginline][endline]*
```

where *D* specifies the device name. # is an optional parameter which specifies the drive number. The filename assigned to the file is specified in *filename*.

beginline and *endline* are optional parameters which specify the first and last line numbers to be stored by LIST. All line numbers with values between *beginline* and *endline* will be stored as well. For example, the following LIST statement would save all program lines between 100 and 500 inclusive on the diskette in drive 2 with the filename PROGA.BAS:

```
LIST "D2:PROGA.BAS", 100, 500
```

The LIST statement saves programs in Atari ASCII format. In this format, every character is assigned an ASCII code.

Loading a Program

The LOAD statement is used to load programs into memory from diskette which were previously saved in tokenized format by the SAVE statement.

The LOAD statement uses the same format as the SAVE statement. The following LOAD statement would load PROGA.BAS from drive 1:

```
LOAD "D:PROGA.BAS"
```

The ENTER statement is used to load a program previously saved in ASCII format with a LIST statement. The following ENTER statement would load PROGB.BAS from drive 1:

```
ENTER "D:PROGB.BAS"
```

*Brackets [] indicate an optional entry.

If the program file specified by ENTER or LOAD is present on the specified drive, it will be loaded into memory. When the loading process has been completed, the READY prompt will be displayed.

If the program specified by ENTER or LOAD is not present on the drive indicated, error 170 (File Not Found) will occur.

The LOAD statement will erase any existing program in memory when the new program is loaded. The ENTER statement merges the program being loaded with any existing program lines in memory. If the program being loaded contains line numbers which duplicate those of the program in memory, the incoming program lines will replace the duplicate lines in memory. Any existing program lines in memory can be erased by entering the NEW statement before executing an ENTER statement.

LOADing and Executing a Program

In the following series of statements, a BASIC program is loaded with the LOAD statement, and then executed with the RUN statement.

```
LOAD "D:PROGA.BAS"  
READY  
RUN
```

This process can be shortened to a single step by including the LOAD statement's parameter with the RUN statement. An example of this usage of the RUN statement is given below.

```
RUN "D:PROGA.BAS"
```

In the preceding example, PROGA.BAS will be executed as soon as it is loaded.

Chaining Programs

The RUN statement can be included as a program line in one program in order to load and execute another program. This process is known as **chaining**.

For example, when the following program is executed, line 500 will cause a second program (PROGB.BAS) to be executed.

```
100 REM PROGA.BAS
200 A = 9: B = 10
300 C = A * B
400 PRINT C
500 RUN "D:PROGB.BAS"
```

When the new program is loaded in line 500, all variable values will be cleared before PROGB.BAS is loaded. This is due to the fact that the RUN statement as used in line 500 executes a LOAD statement. The LOAD statement in turn executes a NEW statement which erases any existing programs in memory and clears all variables.

USING DATA FILES ON DISKETTE

The BASIC statements PRINT# and PUT are used to store data on a diskette. INPUT# and GET are used to read data into RAM from a diskette.

Opening a Disk File

Before a disk file can be used, it must first be opened with the OPEN statement. The OPEN statement will open an input/output channel to the file specified as its parameter. The OPEN statement uses the following configuration:

OPEN #*channel*, *task*, *value*, *D#*:*filename*

The first parameter in the OPEN statement indicates the *channel*. Before an external device can be accessed either for input or output, a channel to the device must have been opened.

The OPEN statement assigns a channel to an external device. Once a channel is assigned, the device can subsequently be accessed via its channel number. If an OPEN statement attempts to open a channel which is already open, an error will occur.

The second parameter in the OPEN statement indicates the activity for which the channel is being opened. The activities which can be specified for the Atari 810 Disk Drive are listed in Table 7-1.

The third parameter in the OPEN statement is ignored when the Atari 810 is specified as the device. A value of 0 should be used.

The final parameter in the OPEN statement consists of the device name for the Atari 810 Disk Drive (D), followed by an optional drive specifier, and the name of the file to be opened.

**Table 7-1. OPEN Statement Task Parameter Values
(Atari 810 Disk Drive)**

<i>Task Parameter Number</i>	<i>Task Description</i>
4	Disk read operation. The file pointer is positioned to the beginning of the file.
6	<p>Disk directory read operation. This operation allows the user to read the disk directory as if it was a data file. The file pointer will be set to the first filename in the directory that matches the specified filename.</p> <p>When the disk directory file is read, the fields containing filenames which match the filename specified in the OPEN statement will be returned. Filenames that do not match will be skipped. The last value that is returned is the number of free sectors.</p> <p>If the following OPEN statement was specified;</p> <p style="text-align: center;">OPEN #1,6,0,"D:*.BAS"</p>

	all files with an extension of .BAS would be returned during a read operation.
8	Disk write operation. The file pointer is positioned to the beginning of the file. Any existing data in the file is erased.
9	Disk write append operation. The file pointer is positioned to the end of the file. The file must already exist or error 170 will occur. The append operation allows data to be added to an existing file.
12	Disk read and write operation. The file pointer is positioned to the beginning of the file, and existing data in the file is not altered. The file must already exist before it can be opened for updating. As data is read or written, the file pointer will be moved forward in the file. Data written to the file will replace existing data.

A file must be opened via the OPEN statement before that file can be referenced in a program. Once a file has been opened, it is referenced with its channel number, not with its filename.

Channels 1 to 5 are always available for use in Atari BASIC programs. Channel 0 is reserved for the editor (E:). The BASIC graphics statements use channel 6. The CLOAD, CSAVE, and LPRINT statements use channel 7. As long as the BASIC program does not use the graphics statements, channel 6 will be available for use. If CLOAD, CSAVE, and LPRINT are not used, channel 7 will be available.

Closing a Data File

Once a program has finished accessing an open file, that file should be closed with the CLOSE statement. This allows that file's

channel number to be assigned elsewhere.

Also, if a file that was open for output is not properly closed, data may be lost. Closing an output file causes any remaining data in the disk drive buffer to be output followed by an end-of-file record. If the open file is not properly closed, the data in the disk drive buffer may be lost.

The following CLOSE statement,

CLOSE #4

will close channel 4.

All open files are closed automatically when an END or RUN statement is executed, or when a program's last statement is executed (only in the program mode). However, it is good programming practice to close any open files with the CLOSE statement.

Writing to a Data File

Once a file has been opened, data can be written to that file using the PRINT# or PUT statements.

The PRINT# statement uses the following configuration when used to output data to a disk file.

PRINT #*channel*; *expression*

channel indicates the channel to be used to send the output. Use the *channel* assigned to the file in the OPEN statement. *expression* consist of one or more string or numeric values to be output. These values are output as ASCII values.

Data is output to the disk drive via the disk drive buffer. The disk drive buffer stores data output from PRINT# statements until the buffer fills or an end-of-line character is encountered.

The disk drive buffer has a capacity of 125 characters. Therefore, data will be output from the buffer to the diskette in 125

character blocks. If an EOL character is output, the disk buffer contents will be sent to the diskette regardless of whether or not the buffer is filled. If a PRINT# statement contains more than one *expression*, these will be concatenated unless separated with an EOL character.

PRINT# statements automatically output an EOL character after outputting the *expressions*. However, a comma or semicolon at the end of a PRINT# statement suppresses the EOL character.

If the previous PRINT# statement *expression* list ended with a semicolon, the current PRINT# statement would output its first character immediately after the final character that was output by the preceding PRINT# statement. If the previous PRINT# statement *expression* list ended with a comma, the current PRINT# statement would begin output at the next column tab stop.

Blank spaces will be placed in the area between the last character output via PRINT# and the first character output at the next tab stop by the current PRINT# statement. Because of the insertion of these blank spaces, it is advisable not to insert commas in PRINT# statements used for disk output.

The PUT statement can also be used to output data to a disk file. PUT takes the following configuration:

PUT #*channel*, *numeric expression*

As with PRINT#, the *channel* specified must be open for output. The value given in *numeric expression* is output to the data file. The value output will lie between 0 and 255. If the value is not an integer, it will be rounded to the nearest integer.

If the value specified does not lie between 0 and 255, it will be output module 256. In other words, 256 would be sent as 0, 257 as 1, 258 as 2, 259 as 3, etc.

Reading From a Data File

The INPUT# and GET statements are used to read data from files

and assign that data to the variables specified in the statement.

INPUT# uses the following configuration:

$$\text{INPUT \#channel} \left\{ \begin{array}{l} ; \\ \end{array} \right\} \text{variable, ...}$$

The *channel* must have been previously opened for input. The *variables* will accept the data values input.

INPUT# will retrieve data from the input device specified. This data will consist of ASCII characters followed by the ASCII end-of-line character. The EOL character will end input to the *variable* specified.

INPUT# will interpret the data being read as either numeric or string--depending on the type of *variables* used as parameters. When a numeric variable is specified, the data being input will be interpreted as numeric data.

The data read via INPUT# will be assigned to the numeric variable indicated until a comma or an EOL character is encountered. Numeric values can be ended either with the EOL character or comma. Commas can not be used to end string values, as they are regarded as part of the string.

If no data is available to be read into the numeric variable, or if the data is invalid, an error will result.

When a string *variable* is specified, the data being input will be interpreted as string data. If no characters are read, the string variable will be assigned the null value. If more characters are read than allowed for in the string variable's DIM statement, the INPUT# statement will disregard the excess characters. The EOL character will end the string input.

The INPUT# statement transfers data from the diskette in 125 character blocks. A single block might contain values which will be assigned to several variables. The variables specified in the INPUT# statement will be assigned the values in the disk buffer in a sequential manner.

If an INPUT# statement attempts to read beyond the end of a disk file, an error will result.

The GET statement is used to read a single numeric value via the open channel specified. GET uses the following configuration:

GET #*channel*, *numeric variable*

The *channel* specified must be open for input. The *numeric variable* indicated will accept the value returned by GET. This value will lie between 0 and 255.

When GET is used with the 810 Disk Drive and the buffer is empty, the initial GET statement will result in a block of data being read into the disk buffer. The first value in the buffer will be assigned to the *numeric variable* specified with the first GET statement.

Each successive GET statement will read a value from the buffer. When the buffer has been emptied, another block of data will be read into the buffer from the diskette.

NOTE and POINT

NOTE and POINT are BASIC statements that aid the user in random access of Atari disk files. Random access is only possible with DOS version 2.0.

The NOTE statement will return the position of the file pointer. The file pointer will be referenced using two separate data items. One data item is the number of the last sector accessed. The other data item is the number of the last character referenced within that sector.

The NOTE statement uses the following configuration:

NOTE #*channel*, *sector*, *character*

The NOTE statement will reference the file opened with the specified *channel* number. The number of the last sector accessed will be assigned to the variable given in *sector*. The

number of the last character accessed within that sector will be assigned to the variable named in *character*.

The sector number returned is the absolute sector number on the diskette. It can be any number from 1 to 719. Remember, a file's sector numbers need not necessarily be sequential. For example, the first sector for a file might be sector 57, the second 147, the third 32, etc.

The POINT statement moves the file pointer to the sector and character number specified. Any subsequent file access will begin at the new file pointer location specified by POINT.

POINT uses the following configuration:

POINT #*channel*, *sector*, *character*

The file open under the *channel* number specified will have its file pointer moved. The file pointer will be moved to the sector number indicated in *sector*, and the character within that sector indicated by *character*. Both *sector* and *character* must be numeric variables. Constants may not be used.

If the file pointer is moved to a sector not assigned to the file opened under the channel number specified in *channel*, one of the following errors will result when an attempt is made to read from or write to that file.

Error 170 (End of File)--Attempted read.
Error 171 (Point Invalid)--Attempted write.

CHAPTER 8. ATARI PRINTERS

Introduction

In this chapter, we will concentrate on sending output to the three Atari printers; the Atari 820 Printer, the Atari 822 Printer, and the Atari 825 Printer.

The process of outputting numbers and text differs only slightly between these three printers. However, the Atari 825 has several programmable features not available on the Atari 820 and 822. These will be described at the end of this chapter.

Another difference between the three Atari printers lies in the width of the lines they output. The Atari 820 and 822 output 40 column lines. This is the same output width as the video display. On the other hand, the Atari 825 generally uses a line width of 80 columns.

One final difference lies in the way in which these three printers are connected to the Atari computer. The Atari 820 and 822 are connected to the Atari computer via the serial I/O port. The Atari 825 printer is connected to the Atari 400 or 800 via the Atari 850 Interface module.

BASIC STATEMENTS FOR PRINTER OUTPUT

Sending output to the printer is generally much the same as sending output to the screen. In the following sections, we will discuss the procedures to output data to the printer.

LIST "P:"

When used with the printer's device name (P:), the LIST statement will output the BASIC program currently in memory

to the printer. One or two optional line numbers may also be used as parameters to output only a portion of the program.

When a program is listed to the Atari 820 or 822, the individual lines will have a maximum width of 40 columns. Program lines greater than 40 characters in length will be continued on the next line. Since the Atari 825 has an 80 column width, only program lines greater than 80 characters will be continued on the next line.

None of the Atari printers can output the graphics characters. On the Atari 820 and 822, graphics characters will appear as blank spaces. On the Atari 825, certain graphics characters cause abnormal output while other graphics characters do not print at all.

The Atari 825 uses certain Atari ASCII codes as printer control characters. If the printer encounters these control characters in a program, peculiar printer output can result. For this reason, control codes should be specified in Atari BASIC programs using the CHR\$ function.

LPRINT

LPRINT outputs data to the printer much as the PRINT statement outputs data to the video display.

LPRINT is designed for use with printers using 40 column output. This presents difficulties in situations where LPRINT is used with the Atari 825. For example, if an LPRINT statement outputs 40 characters or less and ends with a semicolon, or 38 characters or less and ends with a comma, the output from the next LPRINT statement will begin on column 41 of the same line. If a third LPRINT statement follows, output from that statement will begin at the beginning of the next printer line.

If an LPRINT statement outputs over 40 characters on an Atari 825, the next LPRINT statement causes output to begin on a new line--even if the statement ends with a comma or semicolon.

On either the Atari 820, 822, or 825, if an LPRINT statement does not end with either a comma or semicolon, output from the next LPRINT statement will begin at the beginning of the next line.

LPRINT uses channel 7 for printer output. If channel 7 has already been opened for another device, an error will occur when LPRINT is executed. This error will automatically close channel 7, after which LPRINT can be executed.

PRINT# and PUT

Either the PRINT# or PUT statements can be used to send output to the printer. PRINT# and PUT direct output to a channel previously opened via an OPEN statement. If the channel was opened for the printer (device P:), the output for that channel will be directed to the printer.

Characters are output to the printer via PRINT# and PUT in the same manner as they are output to the display screen.

The printer must be powered on when a PRINT# or PUT statement outputs data to the printer. If not, an error will occur.

PRINTER BUFFER

The Atari printers contain enough RAM to hold one line of output. This memory is known as the **printer buffer**. As characters are output to the printer, these are not automatically printed but are instead sent to the printer buffer.

When either an EOL character is encountered or when the buffer fills, an entire line will be output. The buffer will be cleared, and the printer will advance to the next line.

PRINTER CHARACTERS SETS

The Atari printers use a character set that is somewhat different than the character set used by the video display. The Atari printers use the standard ASCII code set to define their character set. The display screen uses the Atari ASCII code set. Both code sets are listed in Appendix C.

Remember that none of the Atari printers are capable of outputting the graphics characters. Also, the Atari 820 and 822 recognize ASCII codes 0 to 31 as a blank space, while the Atari 825 recognizes these as control codes.

Atari 825 Control Characters

The Atari 825 has a number of special printing features such as underlining, double-wide character printing, and condensed character printing. These features are summarized in Table 8-1. These features are activated and deactivated with control characters. These control characters are also listed in Table 8-1.

A control character can be sent to the Atari 825 printer with the LPRINT, PRINT#, or PUT statements just as any other character would be sent.

Control codes can be generated either with the CHR\$ function or via the keyboard. The following program lines would cause a reverse line feed.

```
100 LPRINT CHR$(27);CHR$(10)
```

Control characters can also be generated with the keystrokes listed in Table 8-1. When these keystrokes are entered, the graphics characters given in Table 8-1 will be echoed on the screen. The screen interprets the keyboard entry as an Atari ASCII code and displays the corresponding character on the screen.

The Atari 825 interprets the keyboard entry as a control character. For example, if the following line was entered,

```
100 LPRINT " CTRL-J "      Control-J is
                             entered at the
                             keyboard
```


the paper would advance 1 line in the Atari 825. The graphics character  would be displayed on the video screen.

Table 8-1. Atari 825 Printer Control Characters

Screen Character	Keystrokes	Decimal Code	ASCII Mnemonic	Atari 825 Control Functions
	CTRL-J	10	LF	Line Feed
	ESC/ESC & CTRL-J	27 & 10	ESC LF	Reverse line feed
	ESC/ESC & ESC/CTRL-↑	27 & 28	ESC FS	Half-line feed
	ESC/ESC & ESC/CTRL-←	27 & 30	ESC RS	Reverse half-line feed
	CTRL-M	13	CR	Carriage return
	CTRL-N	14	SO	End underline
	CTRL-O	15	SI	Begin underline
	ESC/ESC & CTRL-A	27 & 01	ESC SOH	1 dot space
	ESC/ESC & CTRL-B	27 & 02	ESC STX	2 dot spaces
	ESC/ESC & CTRL-C	27 & 03	ESC ETX	3 dot spaces
	ESC/ESC & CTRL-D	27 & 04	ESC EOT	4 dot spaces
	ESC/ESC & CTRL-E	27 & 05	ESC ENQ	5 dot spaces
	ESC/ESC & CTRL-F	27 & 06	ESC ACK	6 dot spaces
	CTRL-H	08	BS	Backspace. Code must be followed with a character.
	ESC/ESC & CTRL-N	27 & 14	ESC SO	Begin extended character printing.
	ESC/ESC & CTRL-O	27 & 15	ESC SI	End extended character printing.
	ESC/ESC & CTRL-S	27 & 19	ESC DC3	Standard character spacing. 10 character per square inch.
	ESC/ESC & CTRL-T	27 & 20	ESC DC4	Condensed character spacing. 16.7 characters per square inch.
	ESC/ESC & CTRL-Q	27 & 17	ESC DC1	Proportionally spaced character set.

The preferred method of sending control codes to the Atari 825 is via the CHR\$ function. If a program is listed containing control codes specified by the CHR\$ function, the listing of that program will not affect the printer. However, if a program containing control characters surrounded by quotation marks is listed, those control characters will be heeded by the Atari 825, and the program listing will be affected accordingly.

Line Feed

The line feed code will advance the paper in the Atari 825 by one line or 1/6 of an inch.

Any data received prior to the line feed code will be printed before the paper is advanced. The following statement,

```
100 LPRINT "John";CHR$(10);"William"
```

would generate the following:

```
John
William
```

Reverse Line Feed

The reverse line feed code causes the paper in the printer to be reversed by 1/6 of an inch. The following statement,

```
100 LPRINT "John";CHR$(27);CHR$(10);"William"
```

would result in the following output:

```
William
John
```

Half-line Feed & Reverse Half-Line Feed

The half-line feed control code causes the paper in the printer to be advanced by 1/12 inch.

These control codes are very useful when printing subscripts. For example, the following program line,

```
100 LPRINT "8";CHR$(27);CHR$(28);"10";
    CHR$(27);CHR$(30);"Base Ten"
```

would output:

8₁₀ Base Ten

Carriage Return

When the carriage return code is received, the printer will return to the left margin, and an automatic line feed will be generated. For example, the following program line,

```
100 LPRINT "JOHN";CHR$(13);"WILLIAM"
```

would generate the following output:

JOHN
WILLIAM

Underlining

The SI code causes the underlining of characters to begin. When an SO code is received, the underlining is discontinued. For example, the following program line,

```
100 LPRINT CHR$(15);"UNDERLINE";CHR$(14);" STOP"
```

would generate the following output:

UNDERLINE STOP

Standard, Condensed, and Proportionally Spaced Character Sets

The default character set used on the Atari 825 is the standard character set--where 10 characters per inch are output. By sending the condensed character control code, the condensed character set will be active. In the condensed character set, 16.7 characters per inch are output. The character width is the uniform for all characters within either the standard (10 dot spaces per character) or the condensed (9 dot spaces per character) character set.

In the proportionally spaced character set, characters are assigned different widths. For example, an I would be assigned a more narrow width than a W. However, all digits are assigned uniform widths. Digits will be printed at 12.5 characters per inch. Approximately 14 non-digit characters are output per inch in the proportionally spaced character set.

Condensed and proportionally spaced characters can be mixed in the same output line. However, characters printed with the standard character set cannot be mixed on the same line with condensed or proportionally spaced characters.

The maximum line length on the Atari 825 printer is 8 inches. Therefore, a full line of standard characters would contain 80 characters. A full line of condensed characters would contain 132 characters.

The ESC SO Code causes characters to be printed by the Atari 825 as extended or double width characters. The ESC SI code will end extended character printing. Extended character printing also will end when a carriage return code is encountered.

If double wide characters are being output, 40 standard character set double wide characters would fill a line. 66 condensed double wide characters would fill a line.

Backspace & 1-6 Dot Spaces

The Backspace character code consists of the code BS followed by the number of dot spaces (nn) to be backspaced. The BS code

can appear as either control-H or CHR\$(8). The number of spaces can appear as a CHR\$ function or as a print character or control code. For example,

```
LPRINT CHR$(8);CHR$(100)
```

would backspace by 100 dots.

If nn (number of dot spaces) is specified as a print character or control code, that character code's ASCII decimal equivalent will be used as the number of dots to be backspaced. This number can be from 0 to 127 inclusive.

If d (with an ASCII decimal equivalent of 100) was substituted for CHR\$(100) in our previous example,

```
LPRINT CHR$(8);"d"
```

the result would still be backspacing of 100 dots.

In the standard character set, each character is considered to be 10 dot spaces wide. In the condensed character set, each character is considered to be 9 dot spaces wide. Therefore, BS 10 would backspace one character in the standard character set, and BS 9 would backspace one character in the condensed character set.

In the proportionally spaced character set, the number of dot spaces per character varies from 6 to 18. The number of dot spaces for the width of the 96 ASCII print characters is given in Table 8-2.

The dot spacing control characters as listed in Table 8-1 can be used to add or delete dot spaces between words and/or characters. If dot spaces are added, the line will be extended. By replacing the current number of dot spaces between words and/or characters with a lesser number, the line will be condensed.

Dot spacing can be useful in printing characters in a bold typeface.

**Table 8-2. Dot Space Width of Proportionally Spaced
ASCII Print Character Set**

ASCII Print Character	Decimal Code	No. of Dot Spaces	ASCII Print Character	Decimal Code	No. of Dot Spaces
SP	32	7	O	79	16
!	33	7	P	80	14
"	34	10	Q	81	14
#	35	15	R	82	15
\$	36	12	S	83	12
%	37	16	T	84	14
&	38	14	U	85	15
'	39	7	V	86	16
(40	7	W	87	18
)	41	7	X	88	16
*	42	12	Y	89	16
+	43	12	Z	90	10
,	44	7	[91	12
-	45	12	^	92	12
.	46	7]	93	12
/	47	12	\	94	12
0	48	12	—	95	12
1	49	12	—	96	7
2	50	12	a	97	12
3	51	12	b	98	12
4	52	12	c	99	10
5	53	12	d	100	12
6	54	12	e	101	12
7	55	12	f	102	10
8	56	12	g	103	12
9	57	12	h	104	12
:	58	7	i	105	8
;	59	7	j	106	6
<	60	12	k	107	12
=	61	12	l	108	8
>	62	12	m	109	16
?	63	12	n	110	12
@	64	14	o	111	12
A	65	16	p	112	12
B	66	15	q	113	12
C	67	14	r	114	10
D	68	16	s	115	12
E	69	14	t	116	10
F	70	14	u	117	12
G	71	16	v	118	12
H	72	16	w	119	16
I	73	10	x	120	12
J	74	14	y	121	12
K	75	16	z	122	10
L	76	14	}	123	10
M	77	18		124	7
N	78	16	~	125	10
			~	126	12

CHAPTER 9. ATARI GRAPHICS & SOUND

Introduction

In this chapter, we will provide an overview of the various graphics and sound capabilities that are available in Atari BASIC.

The following commands are used to create graphics in Atari BASIC:

GRAPHICS	LOCATE	PUT/GET
COLOR	PLOT	SETCOLOR
DRAWTO	POSITION	XIO

These commands will be discussed in the following sections.

GRAPHICS

The GRAPHICS command is used to select one of the 9 graphics modes available in Atari BASIC. GRAPHICS is used with the following configuration,

GRAPHICS X

where X is a real numeric constant, variable, or expression which when rounded evaluates to a positive integer.

Generally, the GRAPHICS statement will clear the video display upon execution. However, by adding 32 to the value of X, this display clearing function will be disregarded.

GRAPHICS 0

The characteristics of the various Atari graphics modes are given in Table 9-1.

Table 9-1. Atari Graphics Modes

Graphic Mode Number	Mode Type	No. of Columns	No. of Rows (Split Screen)	No. of Rows (Full Screen)	No. of Colors	RAM Requirement
0	TEXT	40	-	24	1*	993
1	TEXT	20	20	24	5	513
2	TEXT	20	10	12	5	261
3	GRAPHICS	40	20	24	4	273
4	GRAPHICS	80	40	48	2	537
5	GRAPHICS	80	40	48	4	1017
6	GRAPHICS	160	80	96	2	2025
7	GRAPHICS	160	80	96	4	3945
8	GRAPHICS	320	160	192	1*	7900

*1 color; 2 luminances

Mode 0 is the text mode. This is the standard mode that is encountered upon power-on.

Modes 0 contains a 24 by 40 character screen. The left margin is set by default to column 2, and the right margin is set to column 39. These settings allow 38 characters per line. The left and right margins can be altered by POKEing a new location to the memory locations which specify these margins.

The left margin memory address is known as LMARGN and is address 82. The right margin memory address is known as RMARGN and is address 83.

In Graphics Mode 0, only one color is available in the display area. However, 2 different levels of luminance (brightness) are available. The color of the characters will be the same as that of the background. However, the luminance of the characters can differ, making them readable.

Upon start-up in Graphics 0, the display area color is blue (color register 2), and the border area color is black (color register 4). However, by changing the default values of these color registers, the border area and display area color can be altered. This changing of the color registers can be accomplished with the SETCOLOR statement. For example, the following SETCOLOR statement would change the display area from blue to orange.

SETCOLOR 2, 2, 4

The SETCOLOR statement can also be used as shown in the following example to alter the background from black to green.

SETCOLOR 4, 12, 0

The concepts of color registers and the use of SETCOLOR to change color registers will be discussed in the next section.

Color Registers & SETCOLOR

Color registers are memory locations within the Atari which set the foreground, background, and border colors. The Atari contains 5 color registers. These are numbered from 0 through 4 inclusive. The Atari's operating system uses the following RAM addresses to store the current color in each register.

Address Name	Address Location	Color Register No.
COLOR0	Address 708	Color Register 0
COLOR1	Address 709	Color Register 1
COLOR2	Address 710	Color Register 2
COLOR3	Address 711	Color Register 3
COLOR4	Address 712	Color Register 4

Each of the 5 color registers has a default color defined. These default values are listed in Table 9-2.

The default color values for the 5 color registers can be changed with the SETCOLOR command. SETCOLOR uses the following configuration:

SETCOLOR *register#, color#, luminance#*

Table 9-2. SETCOLOR Register Default Values

SETCOLOR Register No.	Default Color No.	Luminance Value	Actual Color
0	2	8	Orange
1	12	10	Green
2	9	4	Dark Blue
3	4	6	Pink
4	0	0	Black

The *register#* indicates the number of the register whose default values are to be altered. The *color#* indicates the color to which the register indicated is to be set. The sixteen available colors are listed in Table 9-3 with their associated SETCOLOR numbers.

The *luminance* indicates the brightness value of the color. Luminance can range from 0 (darkest) to 14 (brightest). Odd luminance values give the same luminance as the next lowest even value. By combining the different color and luminance values, as many as 128 different shades of color can be created.

Table 9-3. Atari Colors and Color Numbers

Color	Color Number	Color	Color Number
Gray	0	Blue	8
Gold	1	Light Blue	9
Orange	2	Turquoise Blue	10
Red	3	Green Blue	11
Pink	4	Green	12
Violet	5	Yellow-Green	13
Blue Purple	6	Orange-Green	14
Blue	7	Orange	15

GRAPHICS 1 & 2

Graphics Modes 1 and 2 are both text modes. The items available for display can be chosen from either of two 64 character sets. The standard character set consists of the upper case letters, digits, and punctuation symbols. The alternate character set consists of the lower case letters and special graphics characters.

The standard character set will be active whenever the Atari is powered on, the System Reset key is pressed, or when a GRAPHICS statement is executed. By executing the following statement,

POKE 756,226

the alternative character set can be selected. If the statement,

POKE 756,224

is subsequently executed, the standard character set will again be active. Table 9-4 lists the characters in the standard and alternative character sets along with their COLOR statement color register values (explained later in this chapter).

































































In Graphics mode 1, the characters are printed at the same height as those specified in Graphics mode 0. However, they are printed at twice the width as the characters in Graphics 0. In Graphics mode 2, the characters are printed at both twice the length and twice the height as the characters in Graphics mode 0.

Graphics modes 1 and 2 use what is known as a split screen display (see Illustration 9-1). The split screen consists primarily of a graphics window with 4 lines of text displayed at the bottom of the display.

In the split screen mode, a PRINT statement will cause data to be displayed in the text window. A PRINT #6 statement causes data to be output to the graphics window.

The split screen can be changed to a full graphics screen by adding 16 to the graphics mode number.

Table 9-4. Standard & Alternative Character Sets in Graphics Modes 1 and 2 and Color Register Values

Character		Value for Color Register			
Std.	Alt.	0	1	2	3
		32*	0	160	128
		33	1	161	129
		34	2	162	130
		35	3	163	131
		36	4	164	132
		37	5	165	133
		38	6	166	134
		39	7	167	135
		40	8	168	136
		41	9	169	137
		42	10	170	138
		43	11	171	139
		44	12	172	140
		45	13	173	141
		46	14	174	142
		47	15	175	143
		48	16	176	144
		49	17	177	145
		50	18	178	146
		51	19	179	147
		52	20	180	148
		53	21	181	149
		54	22	182	150
		55	23	183	151
		56	24	184	152
		57	25	185	153
		58	26	186	154
		59	27	187	**
		60	28	188	156
		61	29	189	157
		62	30	190	158
		63	31	191	159

*155 will designate the same character and color register as 32.

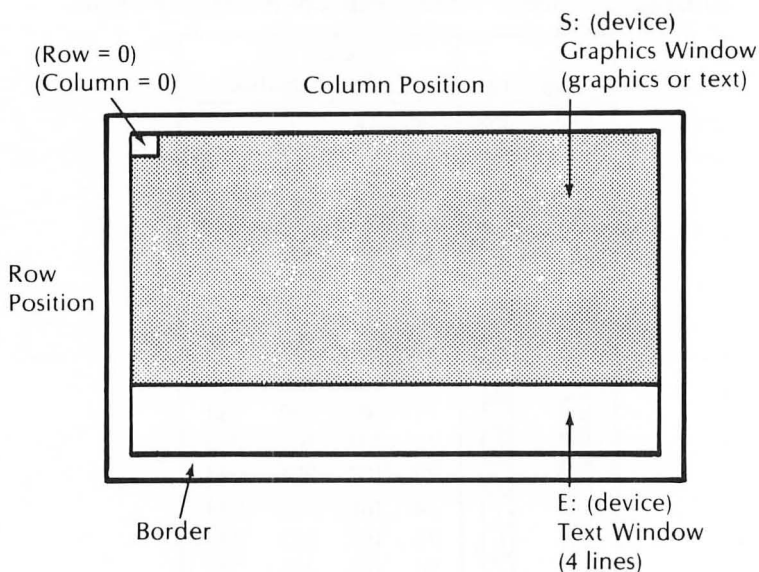
**No value is available to select this color register/character.

Table 9-4 (Cont.). Standard & Alternative Character Sets in Graphics Modes 1 and 2 and Color Register Values

Character		Value for Color Register			
Std.	Alt.	0	1	2	3
@	◆	64	96	192	224
A	a	65	97	193	225
B	b	66	98	194	226
C	c	67	99	195	227
D	d	68	100	196	228
E	e	69	101	197	229
F	f	70	102	198	230
G	g	71	103	199	231
H	h	72	104	200	232
I	i	73	105	201	233
J	j	74	106	202	234
K	k	75	107	203	235
L	l	76	108	204	236
M	m	77	109	205	237
N	n	78	110	206	238
O	o	79	111	207	239
P	p	80	112	208	240
Q	q	81	113	209	241
R	r	82	114	210	242
S	s	83	115	211	243
T	t	84	116	212	244
U	u	85	117	213	245
V	v	86	118	214	246
W	w	87	119	215	247
X	x	88	120	216	248
Y	y	89	121	217	249
Z	z	90	122	218	250
⌈	♠	91	123	219	251
/	⌋	92	124	220	252
⌋	⬆	93	**	221	253
^	⬇	94	126	222	254
□	⬇	95	127	223	255

**No value is available to select this color register/character.

Illustration 9-1. Split Screen Display



Five different default colors are available in graphics modes 1 and 2. These correspond to color registers 0 through 4 (see Table 9-2).

Color register 4 controls the background and border colors. The default color is color 0 with a luminance of 0. This sets the background and border colors to black. The following SETCOLOR statement,

```
SETCOLOR 4,0,4
```

would set the border and background colors to grey in graphics modes 1 and 2. SETCOLOR 4,2,4 would set the background and border colors to orange.

In graphics modes 1 and 2, the color of the characters output in the graphics window depends on the type of character. The color registers and default colors used for each type of character are summarized in Table 9-5.

**Table 9-5. Default Color Registers For Characters
Entered in Graphics Modes 1 and 2**

Type of Character	Color Register	Default Color
Upper Case Alphabetic	0	2 - Orange
Inverse Upper Case Alphabetic	2	9 - Dark Blue
Lower Case Alphabetic	1	12 - Green
Inverse Lower Case Alphabetic	3	4 - Red
Numbers	0	2 - Orange
Inverse Numbers	2	9 - Dark Blue

By executing a SETCOLOR statement, the characters will be output using the colors specified in SETCOLOR. For instance, if the following statement was executed in graphics modes 1 or 2,

```
READY
GRAPHICS
PRINT #6; "9"
```

press ↵ key

press ↵ key a second time

the number 9 would be displayed in dark blue in the graphics window.

If the following SETCOLOR statement was subsequently executed,

```
READY
SETCOLOR 2, 1, 6
```

the number 9 would be changed from dark blue to gold. Subsequent inverse number entries also would be output in gold.

In graphics mode 0, it is not possible to draw lines or plot points. In graphics modes 1 and 2, lines can be drawn with the DRAWTO statement and points plotted with the PLOT statement. The use of these statements will be discussed later in this chapter. First, however, the use of the COLOR statement will be discussed.

COLOR Statement in Graphics Modes 0, 1, and 2

The COLOR statement determines the color register to be used with subsequent PLOT or DRAWTO statements. In graphics modes 0, 1, and 2, COLOR also specifies the character which will be output by subsequent PLOT or DRAWTO statements. COLOR is used with the following configuration,

COLOR *numericexp*

The value of *numericexp* along with the current graphics mode will determine the color register number used.

Table 9-6 can be used to determine the color register number from the COLOR statements *numericexp* value and the current graphics mode. For example, if a COLOR 3 statement was executed in graphics mode 7, SETCOLOR register number 2 would be used for by subsequent PLOT and DRAWTO statements.

In graphics mode 0, the items displayed are characters--not points. In graphics mode 0, the COLOR statement will specify the character to be displayed by subsequent DRAWTO and PLOT statements. Table 9-7 includes the values to be used with the COLOR statement to generate the character desired in graphics mode 0.

For example, the following statement,

```
GRAPHICS 0
COLOR ASC("Y")
PLOT 10,10
```

will display a "Y" in screen position 10,10 in graphics mode 0.

**Table 9-6. Determination of SETCOLOR Register
From COLOR Statement in Graphics Modes 3 Through 8***

Graphics Modes 3, 5, 7

SETCOLOR Register Number	COLOR <i>numericexp</i> Value
0	1
1	2
2	3
3	-
4	0

Graphics Modes 4, 6

SETCOLOR Register Number	COLOR <i>numericexp</i> Value
0	1
1	-
2	-
3	-
4	0










































































































*In graphics modes 0, 1, and 2, the argument of COLOR determines the character to be displayed by PLOT or DRAWTO.

Graphics Mode 8**






SETCOLOR Register Number	COLOR <i>numericexp</i> Value
0	-
1	1
2	0
3	-
4	-

**In graphics mode 8, the color used will always be specified by SETCOLOR register 2. Only the luminance can be altered in graphics mode 8. (See Page 292).

**Table 9-7. Characters Displayed by COLOR
Statement Values in Graphics Mode 0**

Character	COLOR Value Normal/Inverse	Character	COLOR Value Normal/Inverse	Character	COLOR VALUE Normal/Inverse
	0/128		35/163		70/198
	1/129		36/164		71/199
	2/130		37/165		72/200
	3/131		38/166		73/201
	4/132		39/167		74/202
	5/133		40/168		75/203
	6/134		41/169		76/204
	7/135		42/170		77/205
	8/136		43/171		78/206
	9/137		44/172		79/207
	10/138		45/173		80/208
	11/139		46/174		81/209
	12/140		47/175		82/210
	13/141		48/176		83/211
	14/142		49/177		84/212
	15/143		50/178		85/213
	16/144		51/179		86/214
	17/145		52/180		87/215
	18/146		53/181		88/216
	19/147		54/182		89/217
	20/148		55/183		90/218
	21/149		56/184		91/219
	22/150		57/185		92/220
	23/151		58/186		93/221
	24/152		59/187		94/222
	25/153		60/188		95/223
	26/154		61/189		96/224
	27/---		62/190		97/225
	28/156		63/191		98/226
	29/157		64/192		99/227
	30/158		65/193		100/228
	31/159		66/194		101/229
	32/160		67/195		102/230
	33/161		68/196		103/231
	34/162		69/197		104/232

**Table 9-7 (cont.) Characters Displayed by COLOR
Statement Values in Graphics Mode 0**

Character	COLOR Value Normal/Inverse	Character	COLOR Value Normal/Inverse	Character	COLOR Value Normal/Inverse
i	105/233	r	114/242		123/251
j	106/234	s	115/243		124/252
k	107/235	t	116/244	clr scrn	125/---
l	108/236	u	117/245		126/254
m	109/237	v	118/246		127/255
n	110/238	w	119/247	EOL	---/155
o	111/239	x	120/248		---/253
p	112/240	y	121/249		
q	113/241	z	122/250		

The COLOR statement works in much the same fashion in graphics modes 1 and 2. However, there are two important differences in the use of the COLOR statement in graphics modes 1 and 2 as opposed to graphics mode 0.

First of all, two different character sets can be displayed--the standard character set and alternative character set. These are depicted in Table 9-4. The standard character set is active upon power-on, when the System Reset key is pressed, or when a GRAPHICS statement is executed. POKE 756,226 will select the alternate character set, while POKE 756,224 will return to the standard character set.

The second major difference between the use of the COLOR statement in graphics mode 1 and 2 and graphics mode 0 is that in graphics mode 1 and 2, the characters can be produced using SETCOLOR registers 0, 1, 2, or 3. This makes it possible to produce each character using any one of four different colors.

Notice in Table 9-4 that different COLOR statement parameters are used to select the SETCOLOR register.

For example, if the following statements were executed,

```
GRAPHICS 2  
COLOR 159: PLOT 5,5  
DRAWTO 5,0
```

a question mark would be displayed in row 5, column 5 of the graphics window when the PLOT statement is executed. Upon execution of the DRAWTO statement, a column of six question marks will be output from row 5, column 5 on the screen to row 0, column 5.

PLOT

The PLOT statement is used in graphics modes 3 through 8 to display a point on the graphics window. PLOT uses the following configuration,

PLOT *column*, *row*

where *column* gives the column position and *row* gives the row number. The color of the point being plotted will be determined by the color register specified by the most recent COLOR statement. If no COLOR statement had been executed since the computer was powered on, the point will be displayed by the PLOT statement in the background color register.

Although PLOT is generally used in graphics modes 3 through 8, it can also be used in graphics modes 0, 1, and 2. In modes 0, 1, and 2, a character rather than a point is plotted on the screen.

In graphics mode 0, the numeric expression specified with the last COLOR statement will determine the character displayed by PLOT. In graphics modes 1 and 2, the last COLOR statement will choose both the character displayed by PLOT and the color register to display that character. The numeric expression used with COLOR will be 0, if a COLOR statement had not been executed prior to the PLOT statement.

DRAWTO

The DRAWTO statement is used to draw a straight line from the last point displayed by a PLOT statement or another DRAWTO statement to the point given as its argument.

DRAWTO uses the following configuration,

DRAWTO *column, row*

where *column* and *row* specify the screen position where the line is to be drawn to.

The line will be drawn in the color register specified by the last COLOR statement. If no COLOR statement had been previously executed, the background color will be used.

Although DRAWTO is generally used in graphics modes 3 through 8, the statement can also be used in graphics modes 0, 1, and 2. In these modes, the line drawn will consist of characters rather than points.

In graphics modes 1 and 2, the numeric expression specified with the last COLOR statement will determine the character used to compose the line as well as the color register used for the line color. In graphics mode 0, the numeric expression specified with the last COLOR statement will determine just the character used to compose the line.

COLOR 0 will be used by default to determine the character (and color) if a COLOR statement had not been executed since the computer had been turned on.

Graphics Modes 3 Through 8

In graphics modes 3 through 8, points, lines, and solid areas are displayed as opposed to characters. The number of lines on the display screen, number of points per line, point size, and number of color registers available differs among modes 3 through 8. These differences are summarized in Table 9-8.

Graphics modes 3, 5, and 7 normally have split screen displays. However, the display can be changed from split-screen to full screen by adding 16 to the graphic mode number when the GRAPHICS statement is executed.

Graphics modes 3, 5, and 7 each have 4 color registers available (registers 0, 1, 2, and 4). Color register 4 controls the background and border colors. The foreground colors are controlled by registers 0, 1, or 2. Color register 3 is unused in graphics modes 3, 5, and 7.

In graphics mode 3, the split-screen is divided into 20 rows of 40 columns each. In graphics mode 5, the screen is divided into 40 rows of 80 columns each. In graphics mode 7, the screen is divided into 80 rows of 160 columns each.

Table 9-8. Summary of Graphics Modes Features

Mode No.	No. of Columns	No. of Rows		No. of Color Registers	RAM Required
		Split Screen	Full Screen		
0	40	-	24	1	993
1	20	20	24	5	513
2	20	10	12	5	261
3	40	20	24	4	273
4	80	40	48	2	537
5	80	40	48	4	1017
6	160	80	96	2	2025
7	160	80	96	4	3945
8	320	160	192	1	7900

Graphics modes 4 and 6 have one foreground color register and one background and border color register. Even though modes 4 and 6 have the same number of rows and columns as modes 5 and 7 respectively, less RAM is required for modes 4 and 6.

This is due to the fact that modes 4 and 6 are two-color modes, while modes 5 and 7 are four-color modes. One bit per graphics point is required in a two-color mode, while two bits per point are required in a four-color mode. For this reason, approximately one-half as much memory is required in graphics modes 4 and 6 as in modes 5 and 7 respectively.

Graphics mode 8 results in a screen of 160 rows by 320 columns in split screen and 192 rows by 320 columns in full screen. This is the highest resolution available in Atari graphics.

In graphics mode 8, the background/border color register controls the color the graphics points displayed on the screen. Although the foreground color cannot be selected, the luminance of the foreground color register can be set.

Using the COLOR Statement in Graphics Mode 8

In graphics mode 8, the background color as well as all points and lines plotted on it use color register 2 to determine the color. Color 9 (light blue) is the default value in color register 2. The default color can be altered by executing a SETCOLOR statement as shown in the following example.

```
READY  
GRAPHICS 8  
SETCOLOR 2,2,8
```

By executing the preceding SETCOLOR statements, the background color in graphics mode 8 is changed from light blue to orange.

When used in graphics mode 8, the COLOR statement does not determine the color of any points or lines plotted on the screen. Only the luminance of the lines and points is affected.

With a COLOR value of 0, any points or lines will be displayed in color register 2. The color and luminance of the points and lines plotted will match the color and luminance of the background. Therefore, these points and lines will not be visible when they are plotted.

With a color value of 1, any points or lines will be displayed with the luminance value specified in color register 1 and the color in color register 2. In other words, the luminance of the points and lines is affected by the execution of a **COLOR 1** statement, but the color used remains the same as the background color.

The following series of statements illustrate the use of the **COLOR** statement in graphics mode 8.

```
100 GRAPHICS 8
110 COLOR 1
120 PLOT 50,50: DRAWTO 100,100
RUN
```

When the preceding statements are executed, a series of points will be plotted from position 50,50 to 100,100 with a luminance value of 10. These points will be discernable to the naked eye on the video display.

If the following program lines were then added,

```
NEW
100 COLOR 0
110 DRAWTO 60,60
RUN
```

a series of points would be plotted from position 100,100 to 60,60 with a luminance value of 4. Since this is the same luminance as that of color register 2 (which was set by the **COLOR 0** statement), the points plotted would be invisible to the naked eye.

POSITION

Although the graphics cursor is invisible in graphics modes 3 through 8, it can still be moved with the **POSITION** statement. **POSITION** is used with the following configuration,

POSITION *column, row*

where *column* and *row* specify the screen column and row

numbers to which the cursor is to be moved. The next PRINT, PUT, GET, INPUT, or LOCATE statements will use this cursor position. However, DRAWTO, PLOT, and XIO will not use the new cursor position specified by POSITION.

The cursor will not actually move until a subsequent input or output command to the screen has been issued.

LOCATE

The LOCATE command positions the graphics cursor to the position specified and returns the code corresponding to the character or point displayed at that position. LOCATE is used with the following configuration.

LOCATE *column, row, code*

row and *column* specify the screen location. *code* must be a numeric variable. The value returned from the position specified is assigned to the numeric variable named in *code*.

The code returned by LOCATE is interpreted in the same manner as the codes used with the COLOR statement. In graphics mode 0, this code specifies the character being displayed. Use Table 9-7 to interpret the code returned by LOCATE in graphics mode 0.

In graphics modes 1 and 2, the code indicates both the character and the color register used to display that character. Use Table 9-4 to interpret the code returned by LOCATE in graphics modes 1 and 2.

In graphics modes 3 through 8, the code returned by LOCATE identifies the color register in use at the position specified. Use Table 9-6 to interpret the code returned by LOCATE in graphics modes 3 through 8.

When LOCATE is used to read a code from the screen, the cursor will move one location to the right. If the cursor was on the last column of a row when LOCATE was executed, the cursor may

attempt to advance to the first column of the next row resulting in Error 141 (Cursor Out of Range).

LOCATE moves the cursor by altering the values stored in memory address 84 (current cursor row number) and memory addresses 85 and 86 (current cursor column number). The cursor position change as a result of the execution of LOCATE will have no effect on DRAWTO and XIO statements, as they use memory addresses 90, 91, and 92 to determine the next cursor address.

PUT (Graphics)

The PUT statement can be used in the graphics modes to display a character or point. PUT uses the following configuration in graphics.

PUT #6, code

code specifies a character in graphics modes 0 through 2. In graphics modes 3 through 8, *code* specifies a color register.

When PUT outputs data to the screen, memory addresses 84 (next cursor row number) and addresses 85 and 86 (next cursor column number) are incremented. When a subsequent statement that outputs data to the screen or sends data to it is executed, the cursor will advance by one position.

PUT's updating of the next cursor addresses will not affect subsequent DRAWTO or XIO statements as these use different cursor position memory addresses (90, 91, and 92).

If PUT is executed with the cursor in the last column of a row, the cursor will attempt to advance to the first position of the next row. If this occurs, error 141 (Cursor Out of Range) may appear.

XIO (Graphics)

XIO is used in graphics to fill an area of the screen. XIO uses the following configuration.

XIO command, channel #, numexpl, numexp 2, device

command specifies the type of XIO command to be executed. The XIO commands are listed in Table 5-4. The *command* for XIO fill area is 18. The *channel#* specified must be opened for input or output. In graphics, channel 6 is used. *numexpl* and *numexp 2* are given dummy values (0) unless XIO is used in conjunction with an RS-232 operation or to open a channel. *device* refers to the input or output device used with the XIO command. The *device* specified for XIO when used as a fill area command in graphics will be "S:".

The following example illustrates the use of the XIO command to fill an area in graphics.

XIO Example Program

```
100 GRAPHICS 5
200 COLOR 1
300 PLOT 50,20
400 DRAWTO 50,10
500 DRAWTO 10,10
600 POSITION 20,20
700 POKE 765,1
800 XIO 18,#6,0,0,"S:"
```

The following steps must be followed in the order specified in order to fill an area on the graphics display.

1. PLOT the point at the bottom right-hand corner of the figure to be filled. (Reference line 300 in the XIO Example Program).
2. Execute a DRAWTO to the upper right-hand corner of the figure. (Reference line 400 in the XIO Example Program).
3. Execute a DRAWTO to the upper left-hand corner of the figure (Reference line 500 in the XIO Example Program).
4. Execute a POSITION statement to move the cursor to the lower left-hand corner of the figure (Reference line 600 in XIO Example Program).

5. Execute a POKE to address 765. The argument specified with POKE should be set equal to the COLOR statement that was used to plot the points and lines. (Reference line 700 in XIO Example Program).
6. Execute XIO 18,#6,0,0,"S:" and the figure will be filled. (Reference line 800 in the XIO Example Program).

Atari Sound

The Atari's built-in speaker is controlled via memory address 53279. When a 0 is stored at that address, an oscillation is sent to the speaker. By causing the speaker to oscillate a number of times, the speaker will emit a sound. The following program will result in the Atari's speaker emitting sounds.

```

READY
100 FOR I = 1 TO 100
200 POKE 53279,0
300 NEXT I
400 END
RUN

```

Generally, the television set's speaker is used to produce sound rather than the Atari's built-in speaker. In Atari BASIC, the SOUND statement is used to output sound via the television set's speaker.

The SOUND statement is used with the following configuration.

SOUND *voice, pitch, distortion, volume*

Together these four arguments determine the sound produced. *voice* sets one of four voices available with the Atari. These are numbered from 0 to 3. These four voices are independent of each other. In other words, as many as four voices can be sounded at the same time.

pitch sets the pitch of the sound produced by the SOUND statement. The pitch can range from 0 to 255. The highest pitch begins at 0 and the lowest at 255.

The SOUND statement can produce either pure or distorted tones. *distortion* can range between 0 and 15. A *distortion* value of 10 or 14 will produce a pure tone. Any of the other even distortion values (0, 2, 4, 6, 8, 10, and 12) will generate a different amount of noise into the tone produced. The amount of this noise will depend upon the distortion and pitch values specified.

The odd numbered *distortion* values (1, 3, 5, 7, 9, 11, 13) cause the voice indicated in the SOUND statement to be silenced. If the voice is on, an odd-numbered *distortion* value will result in its being shut off.

The *volume* controls the loudness of the voice indicated in SOUND. *volume* ranges from 0 (no sound) to 15 (highest volume).

An Atari BASIC statement with a volume of 0 will turn off the sound. Sound can also be turned off by executing an END, RUN, NEW, DOS, CSAVE, or CLOAD. If the System Reset key is pressed, sound will be turned off. However, if the Break key is pressed, sound will not be turned off.

Appendix A. Atari Error Messages

This appendix describes the error numbers used by the Atari. Error numbers 2 through 21 should only occur when a BASIC program is being run. Error numbers 128 through 173 result from errors in the usage of input/output devices such as disk drives or printers.

Error #	Error Name	Cause & Recovery
2	Insufficient Memory	Additional memory is required to store the statement or to dimension the new string variable. By adding more RAM or by deleting any unused variables, this error can be avoided. This error can also be caused by including a FOR-NEXT statement with too many levels of nesting.
3	Value Error	A numeric value was encountered that was outside of the allowed range i.e. too large or too small. This error can also occur when a negative value is returned when the value should be positive.
4	Too Many Variables	Over 128 variable names have been specified. Any unused names should be deleted.
5	String Length Error	The program attempted to read or write outside of the range for which the string was dimensioned. This also occurs when zero is used as the index. This error can be corrected by increasing the DIM index size.
6	Out of Data Error	The DATA statements did not contain enough data items for the variables in the corresponding READ statements.

Error #	Error Name	Cause & Recovery
7	Line Number Greater Than 32767	The line number is negative or greater than 32767.
8	INPUT Statement Error	An attempt was made to input a non-numeric value into a numeric variable. Be certain that the type of data being entered corresponds to the INPUT variable type.
9	Array or String DIM Error	This error occurs when the program references an array or string which has not been dimensioned. This error also occurs when a DIM statement includes a string or array that was previously dimensioned.
10	Argument Stack Overflow	An expression is too large or there too many GOSUB statements.
11	Floating Point Overflow/Underflow	The program encountered a number with an absolute value less than $1\text{E}-99$ or greater than $1\text{E}+98$. This error also occurs when an attempt is made to divide by zero.
12	Line Not Found	An IF-THEN, ON-GOSUB, ON-GOTO, GOSUB, or GOTO statement referenced a line number that does not exist.
13	No Matching FOR	A NEXT statement was encountered that did not have a corresponding FOR statement.
14	Line Too Long	The line entered is greater than the length of the BASIC line processing buffer length.
15	GOSUB or FOR Line Deleted	A NEXT or FOR statement was encountered for which the corres-

Error #	Error Name	Cause & Recovery
16	RETURN Error	ponding FOR or GOSUB statement had been deleted. A RETURN statement was encountered without a corresponding GOSUB statement.
17	Garbage Error	This error can be caused by faulty RAM or the incorrect use of a POKE statement.
18	Invalid String Character	A string does not begin with a valid character or the argument of a VAL statement is not a numeric string.
19	LOAD Program Too Long	The program being loaded will not fit in the available RAM.
20	Device Number Error	A device number outside of the range 0 to 7 was entered.
21	LOAD File Error	The LOAD statement was incorrectly used to load a program saved by CSAVE or ENTER.
128	BREAK Abort	The Break key was pressed during an I/O operation causing execution to stop.
129	IOCB* Already Open	This error occurs when an attempt is made to use a channel currently in use. Often, the channel causing the error is automatically closed.
130	Nonexistent Device	This error occurs when a program attempts to access a device which is undefined. This error can occur when a filename is given without a required device name (ex. "FILE.-BAS" instead of "D:FILE.BAS").

*IOCB--Input/Output Control Block

Error #	Error Name	Cause & Recovery
131	IOCB Write Only	An attempt was made to read from a file opened only for write operations. The file must be reopened for a read or read/write operation.
132	Invalid Command	This error is generally caused by an illegal command code being used with an XIO or IOCB command.
133	Device/File Not Open	A channel was referenced before it was opened.
134	Bad IOCB Number	An attempt was made to use an illegal IOCB index. A BASIC program can only use channels 1-7.
135	IOCB Read Only Error	An attempt was made to write to a device or file that is opened only for read operations.
136	End of File	The end-of-file record was reached.
137	Truncated Record	This error occurs when an attempt is made to read a record whose record size is larger than the allowed maximum. This error also occurs when an INPUT statement is used to read from a file created with a PUT command.
138	Device Timeout	The external device specified does not respond within the time allowed by the Atari operating system. Be certain the proper device was specified, the device is properly connected, and that the device's power is on.
139	Device NAK	The device does not respond, as it received an incorrect parameter. Check the input/output command

Error #	Error Message	Cause & Recovery
		for any illegal parameters. Also, be certain all cables are properly connected. This error can also result when the Atari 850 Interface Module is unable to accept five, six, or seven bit input at an excessive baud rate.
140	Serial Frame Error	This is a very rare error. If this error reoccurs, have the computer and/or devices checked.
141	Cursor Out of Range	The cursor is outside the defined limits for the current graphics mode. This error can be corrected by using legal cursor positioning parameters.
142	Serial Bus Overrun	This error is due to serial bus data problems. If the error reoccurs, the disk unit, cassette unit, or computer may require service.
143	Checksum Error	The communications on the serial bus are in error. The problem may be due to either defective hardware or faulty software.
144	Device Done Error	This error is generally due to an attempt to write to a write-protected diskette or device.
145	Read. After-write compare Error or Bad Screen Mode Handler	The disk drive identified a difference between what was written and what should have been written. Also, this error can result from a problem with the screen handler.
146	Function Not Implemented	An attempt was made to use a device in a manner not allowed (ex. write to the keyboard).

Error #	Error Message	Cause & Recovery
147	Insufficient RAM	More RAM is required for the graphics mode chosen. Either add RAM or change graphics modes.
150	Port Already Open	An attempt was made to open a serial port already open.
151	Concurrent Mode I/O Not Enabled	Before current mode input/output is enabled with the XIO 40 statement, the serial port must have been opened for concurrent mode.
152	Illegal User Supplied Buffer	Upon the initialization of the concurrent input/output, an incorrect buffer length and address was used.
153	Active Concurrent Mode I/O Error	An attempt was made to access a serial port while another serial port was open and active in the concurrent mode.
154	Concurrent Mode I/O Not Active	The concurrent mode must be active for the input/output operation to be executed.
160	Drive Number Error	The specified drive must be D:, D1:, D2:, D3:, or D4:. This error can also be caused if the drive was not powered on or if a drive buffer was not specified.
161	Too Many Open Files	Another file may not be opened, as the limit of open files has been reached. Generally, only 3 disk files can be open at the same time.
162	Disk Full	All diskette sectors are in use.
163	Unrecoverable System I/O Error	Either the DOS or the diskette contains an error. Try using a different DOS diskette.

Error #	Error Message	Cause & Recovery
164	File Number Mismatch	The POINT statement moved the file pointer to a sector which was not included in the open file. This error can also occur when the file's intra-sector links are incorrect.
165	File Name Error	The filename is illegal. Check the file specification.
166	POINT Data Length Error	The POINT statement attempted to move to a byte number that did not exist within the specified sector.
167	File Locked	An attempt was made to write to, re-name, or erase a locked file.
168	Device Command Invalid	An attempt was made to use an illegal device command.
169	Directory Full	A diskette directory's maximum capacity is 64 filenames.
170	File Not Found	An attempt was made to access a file not present in the disk directory.
171	POINT Invalid	The POINT statement was used with a disk sector in a file not opened for Update.
172	Illegal Append	An attempt was made to open a DOS I file for append using the DOS II operating system. Try copying the DOS I file to a DOS II diskette using DOS II. It is illegal for DOS II to append to DOS I files.
173	Bad Sectors at Format Time	Bad sectors were found while the disk drive attempted to format the diskette. A diskette with bad sectors cannot be formatted. Use another diskette.

Appendix B. Atari BASIC Reserved Words

Reserved Word	Abbrev.	Reserved Word	Abbrev.
ABS		NEXT	N.
ADR		NOT	
AND		NOTE	NO.
ASC		ON	
ATN		OPEN	O.
BYE	B.	OR	
CLOAD	CLOA.	PADDLE	
CHR\$		PEEK	
CLOG		PLOT	PL.
CLOSE	CL.	POINT	P.
CLR		POKE	POK.
COLOR	C.	POP	
COM		POSITION	POS.
CONT	CON.	PRINT	PR. or ?
COS		PTRIG	
CSAVE	CS.	PUT	PU.
DATA	D.	RAD	
DEG	DE.	READ	REA.
DIM	DI.	REM	R. or .
DOS	DO.	RESTORE	RES.
DRAWTO	DR.	RETURN	RET.
END		RND	
ENTER	E.	RUN	RU.
EXP		SAVE	S.
FOR	F.	SETCOLOR	SE.
FRE		SGN	
GET	GE.	SIN	
GOSUB	GOS.	SOUND	SO.
GOTO	G.	SQR	
GRAPHICS	GR.	STATUS	ST.
IF		STEP	
INPUT	I.	STICK	
INT		STRIG	
LEN		STOP	STO.
LET	LE.	STR\$	
LIST	L.	THEN	
LOAD	LO.	TO	
LOCATE	LOC.	TRAP	T.
LOG		USR	
LPRINT	LP.	VAL	
NEW		XIO	X.

Appendix C. Atari ASCII Code Set

In this appendix, the 256 characters in the standard character set of graphics mode 0 are listed along with the Atari ASCII codes for each character. The keystrokes used to produce the characters are also listed along with the associated standard ASCII character (if any). Remember, in graphics modes other than graphics mode 0, an entirely different character may be output.

Some of the Atari ASCII codes produce control characters. When control characters are output using a PRINT statement, nothing is actually displayed on the screen. When control characters are output with a PRINT statement, a control process of some kind will be executed or the cursor will be moved.

Control characters can be included in PRINT statements by supplying the CHR\$ function with the Atari ASCII code of the control character. Control characters can also be output by using an **escape sequence** enclosed within quotation marks.

To produce an escape sequence, first press the Escape key, and then press the keys which will produce the desired control character. For example, if the Escape key is pressed prior to pressing the Control key and the = key, the Atari ASC code 29 for cursor down is produced.

When an escape sequence is used with a control character, the control process does not actually take place during keyboard entry. However, the control character does appear on the screen. When the PRINT statement containing the escape sequence and control character is executed, the control process will take place.

For example, if the following statement was entered,

READY
PRINT "NNN←A" ESC \ CTRL+= pressed here

The output produced would be;

NNA

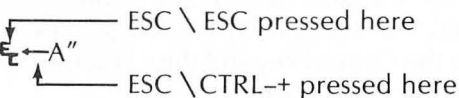
Notice that when the ESC \ CTRL-+ keyboard entry was made, the control process specified (cursor left) did not actually occur. However, the screen character for cursor left (←) was displayed on the screen.

When the PRINT statement was subsequently executed, the cursor left control process did take place. The result of this control process was the movement of the cursor one position to the left (over the third N entry) and the display of the A in place of the previous character entered (N).

If the Atari ASCII code 27 (keyboard entry ESC \ ESC) is included in the PRINT statement just before the control character, that control process will not occur. However, the control character will be displayed.

For example, if the following statement was entered,

PRINT "NNNESC←A"



the following output would be displayed on the screen;

NNN←A

Notice that while the control process did not occur, the control character was displayed.

A great number of the Atari characters can only be entered via the keyboard when the keyboard is in the lower case mode. By pressing the LOWR key once, the keyboard will be in the lower case mode. If the CAPS key is pressed (SHIFT-LOWR keys), the keyboard is returned to the upper case mode.

Atari ASCII Character	ASCII Character	Decimal Code	Keystrokes For Outputting Character
	NULL	0	CTRL-,
	SOH	1	CTRL-A
	STX	2	CTRL-B
	ETX	3	CTRL-C
	EOT	4	CTRL-D
	ENQ	5	CTRL-E
	ACK	6	CTRL-F
	BEL	7	CTRL-G
	BS	8	CTRL-H
	HT	9	CTRL-I
	LF	10	CTRL-J
	VT	11	CTRL-K
	FF	12	CTRL-L
	CR	13	CTRL-M
	SO	14	CTRL-N
	SI	15	CTRL-O
	DLE	16	CTRL-P
	DC1	17	CTRL-Q
	DC2	18	CTRL-R
	DC3	19	CTRL-S
	DC4	20	CTRL-T
	NAK	21	CTRL-U
	SYN	22	CTRL-V
	ETB	23	CTRL-W
	CAN	24	CTRL-X
	EM	25	CTRL-Y
	SUB	26	CTRL-Z
	ESC	27	ESC/ESC
	FS	28	ESC/CTRL--
	GS	29	ESC/CTRL=
	RS	30	ESC CTRL+
	US	31	ESC CTRL*
	Space	32	SPACE BAR
	!	33	SHIFT-1
	"	34	SHIFT-2












Atari ASCII Character	ASCII Character	Decimal Code	Keystrokes For Outputting Character
	#	35	SHIFT-3
	\$	36	SHIFT-4
	%	37	SHIFT-5
	&	38	SHIFT-6
	'	39	SHIFT-7
	(40	SHIFT-9
)	41	SHIFT-0
	*	42	*
	+	43	+
	,	44	,
	-	45	-
	.	46	.
	/	47	/
	0	48	0
	1	49	1
	2	50	2
	3	51	3
	4	52	4
	5	53	5
	6	54	6
	7	55	7
	8	56	8
	9	57	9
	:	58	SHIFT-;
	;	59	;
	<	60	<
	=	61	=
	>	62	>
	?	63	SHIFT-/
	@	64	SHIFT-8
	A	65	A
	B	66	B
	C	67	C
	D	68	D
	E	69	E

Atari ASCII Character	ASCII Character	Decimal Code	Keystrokes For Outputting Character
	F	70	F
	G	71	G
	H	72	H
	I	73	I
	J	74	J
	K	75	K
	L	76	L
	M	77	M
	N	78	N
	O	79	O
	P	80	P
	Q	81	Q
	R	82	R
	S	83	S
	T	84	T
	U	85	U
	V	86	V
	W	87	W
	X	88	X
	Y	89	Y
	Z	90	Z
	[91	SHIFT-;
	\	92	SHIFT-,
]	93	SHIFT-+
	^	94	SHIFT-*
	_	95	SHIFT--
	`	96	CTRL-.
	a	97	(LOWR) A
	b	98	(LOWR) B
	c	99	(LOWR) C
	d	100	(LOWR) D
	e	101	(LOWR) E
	f	102	(LOWR) F
	g	103	(LOWR) G
	h	104	(LOWR) H

Atari ASCII Character	ASCII Character	Decimal Code	Keystrokes For Outputting Character
	i	105	(LOWR) I
	j	106	(LOWR) J
	k	107	(LOWR) K
	l	108	(LOWR) L
	m	109	(LOWR) M
	n	110	(LOWR) N
	o	111	(LOWR) O
	p	112	(LOWR) P
	q	113	(LOWR) Q
	r	114	(LOWR) R
	s	115	(LOWR) S
	t	116	(LOWR) T
	u	117	(LOWR) U
	v	118	(LOWR) V
	w	119	(LOWR) W
	x	120	(LOWR) X
	y	121	(LOWR) Y
	z	122	(LOWR) Z
	{	123	CTRL-;
		124	SHIFT-=
	}	125	ESC/CTRL- < ESC/SHIFT- <
	DEL	126	ESC/BACK S
		127	ESC/TAB
		128	(/A) CTRL-,
		129	(/A) CTRL-A
		130	(/A) CTRL-B
		131	(/A) CTRL-C
		132	(/A) CTRL-D
		133	(/A) CTRL-E
		134	(/A) CTRL-F
		135	(/A) CTRL-G
		136	(/A) CTRL-H
		137	(/A) CTRL-I
		138	(/A) CTRL-J
		139	(/A) CTRL-K

Atari ASCII Character	ASCII Character	Decimal Code	Keystrokes For Outputting Character	Atari ASCII Character	ASCII Character	Decimal Code	Keystrokes For Outputting Character
		140	\backslash CTRL-L			171	\backslash +
		141	\backslash CTRL-M			172	\backslash ,
		142	\backslash CTRL-N			173	\backslash -
		143	\backslash CTRL-O			174	\backslash .
		144	\backslash CTRL-P			175	\backslash /
		145	\backslash CTRL-Q			176	\backslash 0
		146	\backslash CTRL-R			177	\backslash 1
		147	\backslash CTRL-S			178	\backslash 2
		148	\backslash CTRL-T			179	\backslash 3
		149	\backslash CTRL-U			180	\backslash 4
		150	\backslash CTRL-V			181	\backslash 5
		151	\backslash CTRL-W			182	\backslash 6
		152	\backslash CTRL-X			183	\backslash 7
		153	\backslash CTRL-Y			184	\backslash 8
		154	\backslash CTRL-Z			185	\backslash 9
		155	RETURN			186	\backslash SHIFT-;
		156	ESC/SHIFT-BACK S			187	\backslash ';
		157	ESC/SHIFT- >			188	\backslash <
		158	ESC/CTRL-TAB			189	\backslash =
		159	ESC/SHIFT-TAB			190	\backslash >
		160	\backslash SPACE BAR			191	\backslash SHIFT-/
		*161	\backslash SHIFT-1			192	\backslash SHIFT 8
		162	\backslash SHIFT-2			193	\backslash A
		163	\backslash SHIFT-3			194	\backslash B
		164	\backslash SHIFT-4			195	\backslash C
		165	\backslash SHIFT-5			196	\backslash D
		166	\backslash SHIFT-6			197	\backslash E
		167	\backslash SHIFT-7			198	\backslash F
		168	\backslash SHIFT-9			199	\backslash G
		169	\backslash SHIFT-0			200	\backslash H
		170	\backslash *			201	\backslash I

*All Atari ASCII characters from 161-255 inclusive are displayed in reverse.

Atari ASCII Character	ASCII Character	Decimal Code	Keystrokes For Outputting Character	Atari ASCII Character	ASCII Character	Decimal Code	Keystrokes For Outputting Character
J		202	$\backslash \backslash$ J	m		237	$\backslash \backslash$ (LOWR) M
k		203	$\backslash \backslash$ K	n		238	$\backslash \backslash$ (LOWR) N
L		204	$\backslash \backslash$ L	o		239	$\backslash \backslash$ (LOWR) O
M		205	$\backslash \backslash$ M	p		240	$\backslash \backslash$ (LOWR) P
N		206	$\backslash \backslash$ N	q		241	$\backslash \backslash$ (LOWR) Q
O		207	$\backslash \backslash$ O	r		242	$\backslash \backslash$ (LOWR) R
P		208	$\backslash \backslash$ P	s		243	$\backslash \backslash$ (LOWR) S
Q		209	$\backslash \backslash$ Q	t		244	$\backslash \backslash$ (LOWR) T
R		210	$\backslash \backslash$ R	u		245	$\backslash \backslash$ (LOWR) U
S		211	$\backslash \backslash$ S	v		246	$\backslash \backslash$ (LOWR) V
T		212	$\backslash \backslash$ T	w		247	$\backslash \backslash$ (LOWR) W
U		213	$\backslash \backslash$ U	x		248	$\backslash \backslash$ (LOWR) X
V		214	$\backslash \backslash$ V	y		249	$\backslash \backslash$ (LOWR) Y
W		215	$\backslash \backslash$ W	z		250	$\backslash \backslash$ (LOWR) Z
X		216	$\backslash \backslash$ X			251	$\backslash \backslash$ CTRL-;
Y		217	$\backslash \backslash$ Y			252	$\backslash \backslash$ SHIFT-=
Z		218	$\backslash \backslash$ Z			253	ESC/CTRL-2
		219	$\backslash \backslash$ SHIFT-,			254	$\backslash \backslash$ ESC/CTRL-BACK S
		220	$\backslash \backslash$ SHIFT+.			255	$\backslash \backslash$ ESC/CTRL->
		221	$\backslash \backslash$ SHIFT-.				
		222	$\backslash \backslash$ SHIFT*.				
		223	$\backslash \backslash$ SHIFT--.				
		224	$\backslash \backslash$ CTRL-.				
a		225	$\backslash \backslash$ (LOWR) A				
b		226	$\backslash \backslash$ (LOWR) B				
c		227	$\backslash \backslash$ (LOWR) C				
d		228	$\backslash \backslash$ (LOWR) D				
e		229	$\backslash \backslash$ (LOWR) E				
f		230	$\backslash \backslash$ (LOWR) F				
g		231	$\backslash \backslash$ (LOWR) G				
h		232	$\backslash \backslash$ (LOWR) H				
i		233	$\backslash \backslash$ (LOWR) I				
j		234	$\backslash \backslash$ (LOWR) J				
k		235	$\backslash \backslash$ (LOWR) K				
l		236	$\backslash \backslash$ (LOWR) L				

All Atari ASCII characters from 161-255 inclusive are displayed in reverse.

Appendix D. Atari 400/800 Memory Map

The following illustrations and tables depict the organization of memory in the Atari 400/800. Note that the addresses for the top of RAM, OS, and BASIC may differ according to the amount of memory present.

Illustration D-1. Memory Map Without BASIC

65536	Operating System ROM ¹
57344	Floating Point Routines
55296	Hardware Registers ²
53248	Not Used
49152	Cartridge Slot A
40960	Cartridge Slot B ³
32768	RAM (8-40K)
10879	DOS
1792	Operating System RAM ⁴
0	

Illustration D-2. Memory Map With Atari BASIC

65536	Operating System ROM ¹
57344	Floating Point Routines
55296	Hardware Registers ²
53248	Not Used
49152	BASIC ROM ⁵
40960	Cartridge Slot B ³
32768	BASIC Program Area
10879	DOS
1792	Operating System RAM ⁶ & 8K BASIC
0	

- ¹ Reference Table D-1.
- ² Reference Table D-2.
- ³ Atari 800 only.
- ⁴ Reference Table D-3.
- ⁵ Reference Table D-4.
- ⁶ Reference Table D-5.

Table D-1. Operating System ROM Memory Addresses

Memory Address	Reference
62436-65535	Display & Keyboard Handling Routines
61667-62435	Monitor
61249-61666	Cassette Handling Routines
61048-61248	Printer Handling Routines
60906-61047	Disk Handling Routines
59716-60905	SIO
59093-59715	Interrupt Handling Routines
58534-59092	CIO
58496-58533	Initial RAM Vectors
58448-58495	Jump Vectors
58432-58447	Cassette Vectors
58416-58431	Printer Vectors
58400-58415	Keyboard Vectors
58384-58399	Screen Vectors
58368-58383	Editor Vectors
57344-58367	Character Set

} Operating
System
Vectors

Table D-2. Hardware Register Memory Addresses

Memory Address	Reference
54784-55295	Not Used
54272-54783	ANTIC
54016-54271	PIA
53760-54015	POKEY
53504-53759	Not Used
53248-53503	CTIA or GTIA

Table D-3. Operating System RAM Memory Addresses

Memory Address	Reference
1152-1791	User RAM
512-1151	Operating System RAM (detailed below)
1021-1151	Cassette Buffer
1000-1020	Spare
960-999	Printer Buffer
944-959	I/O Channel 7
928-943	I/O Channel 6
912-927	I/O Channel 5
896-911	I/O Channel 4
880-895	I/O Channel 3
864-879	I/O Channel 2
848-863	I/O Channel 1
832-847	I/O Channel 0
794-831	Handler Address Tables
780-793	Miscellaneous
768-779	DCB
736-767	Miscellaneous
712-735	Spare
704-711	Colors
656-703	Screen RAM
648-655	Miscellaneous
624-647	Game Controllers
554-623	Miscellaneous
512-553	Interrupt Vectors
256-511	Stack
128-255	User Page 0 RAM
0-127	Operating System Page 0 RAM

Table D-4. BASIC ROM Memory Addresses

Memory Addresses	Reference
48549-49151	Floating Point
47733-48548	I/O Routines
47543-47732	Graphics
47382-47542	Errors
47128-47381	CONT Subroutines
45321-47127	Execute Statement
45002-45320	Execute Function
44164-45001	Execute Operator
44095-44163	Operator Precedence
43744-44094	Execute Expression
43632-43743	Operator Table
43520-43631	Statement Table
43359-43519	Execute CONT
43135-43358	Memory Manager
42509-43134	Syntax Tables
42159-42508	Statement Name Table
42082-42158	Search
41056-42081	Syntax
41037-41055	Warm Start
40960-41036	Cold Start

Table D-5. Operating System RAM and BASIC

Memory Address	Reference
End of Free RAM-1792	BASIC Program
1536-1791	Free RAM
1406-1535	Input Line Buffer
1152-1405	Syntax Stack
512-1151	Operating System RAM
256-511	Stack
128-255	BASIC Page 0 RAM
0-127	Operating System Page 0 RAM

Appendix E. Atari PEEK and POKE Locations

This appendix lists memory addresses that BASIC programmers may wish to access via the PEEK or POKE statements.

In BASIC, memory addresses as well as the contents at those addresses are given in decimal notation. Each address contains a value between 0 and 255.

Two consecutive addresses are required to store numbers greater than 256. In these instances, the value of the first address plus the value of the second address multiplied by 256 will result in the total value. For example, $\text{PEEK}(97) + 256 * \text{PEEK}(98)$ will return the ending graphics cursor column.

Most Atari memory locations are referred to by name as well as by decimal memory address. Both are given in Appendix E.

Decimal Address	Name	Description
		Memory Addresses
14,15	APPMHI	These addresses contain the highest address that can be used for program lines and variables.
88,89	SAVMSC	These addresses contain the lowest screen memory address. The contents of that address will be displayed in the screen's upper right-hand corner.
128,129	LOMEM	The BASIC low memory pointer.
144,145	MEMTOP	The BASIC top of memory pointer.

741,742	MEMTOP	The highest address in the free memory address will be returned by $PEEK(741) + PEEK(742) * 256 - 1$.
743,744	MEMLO	These locations contain the lowest address in the free memory area.

Screen Addresses		
82	LMARGIN	This address gives the column position of the left margin in graphics 0 mode. The default value is 0.
83	RMARGIN	This address gives the column position of the right margin of the screen in graphics 0 mode. The default value is 39.
84	ROWCRS	This address gives the current row position.
85,86	COLCRS	This address gives the current column position.
87	DINDEX	This address gives the current screen mode.
90	OLDROW	This address specifies the starting graphics cursor row for DRAWTO and XIO18 statements.
91,92	OLDCOL	This address gives the beginning graphics cursor column for DRAWTO and XIO 18 statements.

93	OLDCHR	This address contains the character beneath the cursor. This value will be used to redisplay the character when the cursor is moved.
94,95	OLDADR	This address contains the current text cursor address. This value is used with address 93 to restore the character beneath the cursor once the cursor is moved.
96	NEWROW	This address contains the ending cursor row for a DRAWTO or graphics XIO statement.
97,98	NEWCOL	This address contains the ending cursor column for a DRAWTO or graphics XIO statement.
201	PTABW	This address indicates the number of columns between tab stops. The default value is 10.
656	TXTROW	This address indicates the cursor row in the text window. This value will range from 0 to 3, with 0 indicating the top row in the text window.

657,658	TXTCOL	This address indicates the cursor column in the text window. This value will range from 0 to 39, with 0 being the first column.
752	CRSINH	A value of 0 at this address results in the cursor not being visible. Any other value results in the cursor being visible.
755	CHACT	This address generally has a value of 2. Any other value will result in the cursor's being opaque, the cursor being absent, or characters being inverted. These values and their effect are summarized in Table E-1.
756	CHBAS	This address indicates the character set to be used in graphics modes 1 and 2 (224 = standard; 226 = alternate).
763	ATACHR	This address contains the Atari ASCII code for the last character read or written or last graphics point output.
765	FILDAT	The address contains the fill data to be used with a graphics XIO command.

Graphics Addresses		
708	COLOR0	Color register 0.

709	COLOR1	Color register 1.
710	COLOR2	Color register 2.
711	COLOR3	Color register 3.
712	COLOR4	Color register 4.

		Cassette Buffer
61	BPTR	This address contains a pointer to the next location to be accessed in the cassetted buffer.
63	FEOF	If this address contains a 0, an end-of-file has not been encountered. A value of 0 indicates an end-of-file has been encountered.
649	WMODE	This address indicates the present cassette operation (0 = read; 128 = write).
650	BLIM	This address indicates the size in bytes of the cassette buffer (0-128).
1021-1151	CASBUF	These addresses are used as the cassette buffer.

		Printer Addresses
29	PBPNT	This address contains a pointer to the current location in the printer buffer.

30	PBUFSZ	This address indicates the size of the printer buffer (40 = normal mode; 29 = side-ways mode).
960-999	PRNBUF	These addresses are available for the printer buffer.

		Keyboard Addresses
17	BRKKEY	This address indicates that the Break key has been pressed (0 indicates Break pressed).
694	INVFLG	This address controls whether keyboard entries result in normal or inverse video character output (0 = normal; non-zero = inverse).
702	SHFLOK	This address indicates whether the caps or control locks are in effect (0 = normal--no locks; 64 = caps lock; 128 = control lock).
764	CH	This address contains the value of the key which was previously pressed. If no key was pressed, the address will contain 255.
53279	CONSOL	Executing a PEEK to this location returns a value which indicates whether a special function key has been pressed. These values along with the function key indicated are listed in Table E-2.

		POKE (53279,8) retracts the core of the built-in speaker while POKE (53279,0) extends it. When these two statements are alternated, clicking sounds will be emitted from the speaker.
--	--	---

		Miscellaneous
65	SOUNDR	If the value for this address is 0, sound can be heard over the television set during disk or cassette accessing. A value of 0 eliminates this sound.
186,187	STOPLN	These addresses return the line number where execution of a BASIC program was stopped due to a STOP statement, a TRAP statement, an error, or the Break key being pressed.
195	ERRSAV	This address contains the error number if an error takes place.
212,213	FR0	These addresses contain a value which is to be returned to a BASIC program from a USR function.
251	RADFLG or DEGFLG	This address determines whether trigonometric functions are calculated using degrees or radians (0 = radians; 6 = degrees).

**Table E-1. Address 755 Values and Effects on
Cursor and Character Display**

Address 755 Value	Cursor Visible/Not Visible	Cursor Transparent/Opaque	Characters Normal/Inverse
0	Not Visible	Transparent	Normal
1	Not Visible	Opaque	Normal
2	Visible	Transparent	Normal
3	Visible	Opaque	Normal
4	Not Visible	Transparent	Inverted
5	Not Visible	Opaque	Inverted
6	Visible	Transparent	Inverted
7	Visible	Opaque	Inverted

Table E-2. PEEK (53279) Function Key Values

Value Returned	Function Keys Pressed
0	OPTION, SELECT, & START
1	OPTION & SELECT
2	OPTION & START
3	OPTION
4	SELECT & START
5	SELECT
6	START
7	None

INDEX

- ABS 92
- AC Power Adapter 10, 11
- ADR 92
- Alternative Character Set 276
- AND 56, 57, 58, 92, 93, 94
- Append Operation 225
- Argument 74
- Arithmetic Expressions 51, 54
- Arithmetic Operators 53
- Arrays 49, 50, 51
- Arrow Keys 34
- ASC Function 78, 94
- ASCII 75
- ASCII Code Set 295, 297, 298, 299, 300
- Assignment Statements 60
- Atari 400 7, 8, 9
- Atari 400, Installation 21
- Atari 410 Program Recorder 9, 14, 15, 147, 163, 189
- Atari 800 7, 8, 9
- Atari 800, Installation 19
- Atari 810, Installation 21, 22, 23
- Atari 810 Disk Drive 10, 15, 16, 148, 163, 205
- Atari 820 Printer 10, 11, 149, 164, 253
- Atari 820 Printer, Installation 23
- Atari 822 Printer 16, 149, 164, 253
- Atari 822 Printer, Installation 24
- Atari 825 Printer 16, 17, 149, 164, 253
- Atari 825 Printer, Installation 24, 25
- Atari 830 Modem 17
- Atari 850 Interface Module 16, 17, 24, 25, 152, 165
- Atari ASCII 75, 78
- Atari ASCII Code Set 295, 297, 298, 299, 300
- Atari ASCII Format 192, 193
- Atari BASIC 9, 37
- Atari Educational System 9
- Atari Key (⌘) 34
- Atari Keyboard 11, 148
- ATN 94, 95
- Auto Repeat Key 36
- AUTORUN.SYS 215
- Back S Key 35
- Backspace, Printer 260
- BASIC, Atari 9, 37
- BASIC Reserved Words 294
- BASIC ROM Cartridge
- Binary Load 236, 237
- Binary Save 234, 235, 236
- Blocks 192, 197, 201
- Boolean Expression 51
- Boolean Operations 53, 56, 57
- Booting, DOS 217, 218
- Branching Statements 68, 69
- Break Key 32, 72, 73
- Buffer, Cassette 198
- Buffer, Disk 217, 247, 248, 250
- BYE 95
- Bytes 13
- C: 147, 163, 167
- Caps Key 33, 296
- Caps/Lowr Key 33
- Carriage Return 259
- Carriage Return/Line Feed 63

316 User's Handbook to the Atari 400/800 Computers

- Cartridges, ROM 25
- Cassette Buffer 198, 201
- Cassette Tape 189
- Chaining 197, 243
- Channel 145, 162, 166, 169
- Channel, I/O 145
- Channel Switch 10
- CHR\$ 78, 96, 254, 258, 261
- Clear Key 35
- CLOAD 83, 95, 96, 104, 174, 190, 192
193, 194
- CLOG 96
- CLOSE 97, 198, 200, 246, 247
- CLR 97
- COLOR 98, 99, 100, 101, 102, 137, 156,
267, 272, 273, 274, 275, 276, 277, 278
280, 281
- Color Registers 100, 265, 26, 272
- COM 102, 103
- Comma, Formatting 63
- Compound Expressions 52
- Concatenation 77
- Condensed Character Set 260
- Conditional Statements 68
- Constants 47
- CONT Command 32, 72, 73, 103, 180
- Control Characters 256, 295, 296
- Control Key 33
- Controller Jacks 10
- Copy File 223, 225, 226, 227
- COS 104
- Create MEM.SAV 238, 239
- CSAVE 83, 95, 104, 190, 191, 192,
193, 194
- Cursor Control 86

- D: 147, 148, 149
- DATA 60, 61, 62, 105, 106, 170, 171, 172
- Data blocks 198
- Data Files 189, 190, 244
- Default 222
- DEG 106, 107, 169
- Delete File 227, 228
- Delete Key 35
- Delimiter 61
- Density 15
- Device Names 146, 147, 199
- Device Timeout Error 147

- DIM 49, 50, 51, 102, 107, 108, 109
- Directory Full Error 224
- Disk Buffer 217, 247, 248, 250
- Disk Directory 220
- Disk Drive 10, 15, 16
- Disk Files 213
- Disk Full Error 224
- Diskettes 205, 207, 208
- Diskettes, Double-Sided 15, 211
- Diskettes, Single-Sided 15, 211
- Display Lines 41
- DOS 110, 111, 215, 216, 218
- DOS 1.0 110, 111, 215, 216, 218
- DOS 2.0 110, 111, 215, 216, 218
- DOS Menu 110, 111, 218, 219, 220
- DOS.SYS 111, 216
- Dot Spaces, Printer 260, 261, 262
- Double-Sided Diskettes 211
- Down Arrow Key 34
- DRAWTO 99, 112, 113, 138, 139, 151
161, 272, 278, 283
- Duplicate Diskette 233, 234
- Duplicate File 239, 240, 241

- E: 147, 150
- Editor 149, 150
- ELSE 68
- END 40, 73, 74, 114
- END Parameter 235
- ENTER 114, 115, 134, 136, 174, 190, 192
193, 194, 242, 243
- EOF Record 198
- EOL Character 84, 85, 201, 202, 203, 248,
249, 255
- Error Messages 43, 287, 288, 289, 290
291, 292, 293
- Error Traps 91
- ESC Key 36, 79
- Escape Sequence 36, 79, 80, 86, 295, 296
- Execution, Program 40, 41
- EXP 115, 116
- Exponentiation 54
- Expressions 51, 52

- Fields 189, 190
- File Management Subsystem 211
- File Not Found Error 224
- Filename Extension 213, 214

- Filename Match Characters 213, 214
- Filenames 213
- Files 189, 213
- Floating Point Decimal 44
- Floppy Diskettes 207, 208
- FOR 66, 67, 68, 116, 117, 118, 141, 142
- Format Diskette 232
- Formatting 232
- FRE 118
- From Paramter 223
- Functions 74

- Game Controls 17
- GET 119, 120, 121, 122, 123, 197, 200, 202, 203, 244, 248, 250
- GOSUB 69, 70, 71, 72, 123, 124, 159, 160, 172
- GOTO 42, 69, 70, 125, 126
- GRAPHICS 101, 126, 137, 151, 263
- Graphics Characters 80, 81, 82
- Graphics Mode 0 263, 264, 275
- Graphics Mode 1 267, 268, 269, 270, 271, 273
- Graphics Mode 2 267, 268, 269, 270, 271, 273
- Graphics Modes 3,5,7 273, 278, 279, 280
- Graphics Modes 4,6 273, 278, 279, 280
- Graphics Mode 8 274, 278, 280
- Graphics Window 270

- Half-Line Feed 259
- Hard Disks 205, 206
- Hard Sectors 209, 210
- Hardware Stack 183, 184
- Home 87
- I/O Channel 145, 162, 166, 199, 246
- I/O Operations 145, 146, 245, 246
- IF THEN 68, 90, 126, 127, 128
- Immediate Mode 37
- Index Hole 209, 210
- Index Variable 67
- INIT Address 235, 236
- INPUT 64, 65, 129, 130, 131, 132
- INPUT# 197, 200, 202, 203, 244, 248, 249, 250
- Input Error Checks 89, 90
- Input Programming 88

- INPUT Prompt 65, 66
- Insert Key 35
- INT 132, 133
- Integers 44
- Interpreters 18

- Joysticks 17, 178, 179

- K: 147, 148
- Keyboard, Atari 11, 20
- Keyboard Controllers 17
- Keywords 41, 42, 148
- Kilobytes 14

- Languages 18
- Leader 198
- Left Arrow Key 34
- LEN 77, 133
- LET 47, 60, 133, 134
- Line Feed 258
- Line Numbers 38, 39
- LIST 40, 42, 114, 134, 135, 136, 190, 191, 192, 193, 194, 240, 242
- List File 221, 222
- LIST P: 253
- Listing, Program 42
- LOAD 136, 137, 190, 192, 193, 194, 242, 243
- Loading 189
- LOCATE 137, 138, 139, 282, 283
- Lock File 230, 231
- LOG 139
- Logical Operators 56, 57
- Loops 66, 67
- Loops, Nested 67
- LOWR Key 33, 296
- LPRINT 63, 64, 139, 140, 145, 254, 255

- Megabyte 205
- Memo Pad Mode 25, 110
- Memory Addresses 301 through 313
- MEM:SAV 111, 215, 224, 238, 239
- Merging 224
- Modem 17
- Modulo 202, 248
- Monitor 11, 12
- Monitor, Installation 13
- Monochromatic Text Mode 12

318 User's Handbook to the Atari 400/800 Computers

- Nested Loops 67
- NEW 40, 83, 136, 140, 141, 143, 193
- NEXT 66, 67, 68, 116, 117, 118, 141, 142
- NOT 56, 57, 58, 142
- NOTE 143, 216, 250
- Numeric Data 43, 44

- ON GOSUB 72, 90, 124, 144, 159, 172
- ON GOTO 70, 90, 125, 144, 145
- OPEN 97, 119, 120, 121, 122, 129, 130, 131, 132, 145, 146, 147, 148, 149, 150, 151, 152, 163, 164, 165, 166, 198, 199, 244, 245, 246
- Operands 51
- Operating System 9, 17, 18
- Operators 51
- Option Key 31
- OR 56, 57, 58, 153, 154
- Order of Evaluation 53

- P: 147, 253
- PADDLE Function 154, 155
- Paddles 17, 154, 155
- Parameters 41
- PEEK 83, 87, 155
- Pixel 101
- PLOT 98, 101, 102, 112, 113, 151, 156, 157, 161, 272, 277, 278
- Plug-In Cartridges 13
- POINT 157, 158, 215, 250, 251
- POKE 83, 84, 88, 155, 158, 159, 276
- POP 159, 160, 172
- POSITION 84, 87, 160, 161, 281, 282
- Power On 26, 27, 28, 29
- PRINT 42, 62, 63, 84, 161, 162, 163, 164, 165, 295, 296
- PRINT# 197, 200, 201, 244, 247, 248, 255
- Print Zone 63
- Printer Buffer 255
- Printer Character Sets 255, 260
- Printer Control Characters 256, 257, 258
- Program Execution 41
- Program Files 190
- Program Lines 41
- Program Listing 42
- Program Mode 37, 38
- Program Recorder 9, 14, 15
- Program Recording Formats 191
- Programs, Applications 18
- Prompt Messages 65, 66, 88, 89
- Proportional Character Set 260, 262
- PTRIG 166
- PUT 166, 167, 168, 169, 197, 200, 202, 244, 247, 248, 255, 283

- R: 152
- RAD 169
- RAM 14, 83
- Random Access 205
- READ 60, 61, 62, 105, 106, 109, 170, 171
- Read/Write Head 205
- Reading 189, 248
- READY Message 29, 37
- Records 189, 190, 191
- Relational Expression 51
- Relational Operations 53, 55
- REM 59, 60, 171
- Remark Statements 59
- Rename File 228, 229, 230
- Reserved Words 41, 294
- RESTORE 61, 62, 170, 171, 172
- RETURN 71, 123, 172
- Return Key 32
- Reverse Half-Line Feed 259
- Reverse Line Feed 258
- Right Arrow Key 34
- RND 172, 173
- ROM 14
- ROM Cartridges, Installation 25, 26
- Rounding 46
- RUN 37, 173, 174, 243
- RUN Address 235, 236
- Run At Address 237, 238
- RUN C: 195, 196, 197
- Run Cartridge 22, 223

- S: 147, 150, 164
- SAVE 95, 137, 173, 174, 190, 191, 240, 241
- SAVE C: 174, 191
- Saving Programs 189, 190, 240
- Scientific Notation 45, 46
- Screen I/O Operations 151
- Screen Margins 88
- Search Spec 221, 222

- Sectors 208, 209
- Select Key 32
- Semicolon, Formatting 63
- SETCOLOR 100, 101, 175, 265, 266, 271, 273, 280
- Sequential Access 205
- SGN 175
- Shift Key 32
- Simple Expression 52
- SIN 175, 176
- Single-Sided Diskettes 211
- Soft Sectors 209, 210, 211
- Software 17
- SOUND 176, 177, 285, 286
- SQR 74, 177
- Standard Character Sets 260, 276
- Start Key 31
- START Parameter 235
- Statement 41
- STATUS 177
- STATUS Codes 178
- STEP 67, 141, 142
- STICK 178
- STOP 73, 180
- STR\$ 181
- STRIG 179
- String Concatenation 77
- String Handling 76
- String Variables 49, 50
- Strings 43
- Subroutines 70, 71
- Subscript 49, 50
- Subscripted Variables 49, 50
- Substrings 76
- System Reset Key 31, 72, 73
- TAB Function 85
- Tab Key 35, 85
- Tab Stops 85, 86
- Tables 49, 50
- Television Set, Display 11, 12
- Television Set, Installation 11, 12
- Text Data 43
- TO Parameter 223, 224
- Tokenized Format 192
- Tracks 208, 209
- TRAP 91, 181, 182
- TV Switch Box 9, 11, 19, 20
- Unary Operation 54
- Underlining, Printer 259
- Unlock File 231
- Up Arrow 34
- USR 182,
- VAL 183, 184
- Variable Names 48
- Variable Name Table 82, 83, 193, 194
- Variable Storage 82
- Variables 47
- Variables, String 48, 49
- Video Display 11
- Winchester Disk 205, 206
- Write DOS 213, 232
- Writing 189
- XIO 138, 139, 161, 184, 185, 187, 188, 283, 284, 285

\$13.95

USER'S HANDBOOK TO THE ATARI 400/800® COMPUTERS

The User's Handbook to the Atari 400/800® Computers is a clear, concise, and practical guide to the capabilities and operation of the Atari 400 and 800 computers, as well as the various Atari peripherals and expansion devices.

A complete description of the Atari 400 and 800 computers, Atari 410 Program Recorder, Atari 810 Disk Drive, Atari 850 Interface Module, and the Atari 820, 822, and 825 Printers is included. A step-by-step guide to the set-up, operation, maintenance, and programming of the Atari 400 and 800 is also offered.

The following topics are covered in detail:

- Atari Installation
- Plug-In Cartridges
- Atari Power-On Sequence
- Atari Keyboard Usage
- Atari BASIC Programming
- Using Atari DOS
- Atari 410 Program Recorder Usage
- Atari 810 Disk Drive Usage
- Atari Printer Usage
- Atari Graphics
- Atari Sound
- Atari BASIC Reference Guide

No user or potential user of the 400 or 800 computer should be without the User's Handbook to Atari 400/800 Computers.



LC: 82-051088

ISBN: 0-938862-15-4