ATARI® XE™
User's Handbook

Weber Systems, Inc. Staff

Atari XE™
User's Handbook

Atari XE™ User's Handbook

by
WSI Staff

Weber Systems, Inc.
Cleveland, Ohio

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

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

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

Atari XE™ User's Handbook

Library of Congress Cataloging in Publication
Main entry under title:

Atari XE User's Handbook

Includes index
1. Atari XE computers   I. Weber Systems, Inc.
QA76.8.A828A83 1985   005.265   85-13896
ISBN 0-938862-41-3

Typesetting and Layout: Tina Koran and Susan Zaksheske

Contents

Introduction & Acknowledgements

Atari XE User's Handbook is meant to serve as a tutorial as well as on-going reference guide to the operation and programming of the Atari XE computers. All of the Atari's important features are discussed. These include the following:

- Installation
- Keyboard usage
- BASIC programming
- Graphics
- File handling
- DOS usage

A number of examples are included with the text to illustrate the topics being discussed. Terms that may be unfamiliar to the reader will be presented in bold in the text. These terms will be defined in subsequent paragraphs.

Chapter 1 of this book is meant to serve as an introduction to the XE computers and their peripherals. Topics include the system unit, the 1050 disk drive, the 1010 cassette unit, 6502C CPU, ANTIC, POKEY, DOS 2.5, operating system, Atari BASIC and peripherals such as printers, joysticks, and modems.

Chapter 2 details the installation procedure for the Atari XE as well as its start-up. Keyboard usage and the connecting of peripherals are also discussed here.

Chapter 3 is meant to serve as an introduction to programming the 130XE in Atari BASIC. The following topics are discussed:

- BASIC start-up
- Program entry
- Listing a program
- Editing a program
- Running a program
- Saving and loading a program
- Data types
- Operators
- Variables

Chapter 4 discusses additional fundamental programming concepts. These include the following:

- Input and output
- Tables and arrays
- Functions
- String handling
- Program concatenation

Chapter 5 discusses the use of files in Atari BASIC. Both sequential and random file access are covered in detail.

Chapter 6 describes techniques for outputting graphics using BASIC commands. The video game, BARACADE, will be designed in this chapter using the Atari's advanced graphics and sound capabilities.

Chapter 7 consists of a detailed discussion of DOS 2.5. Topics covered include:

- DOS start-up
- DOS keyboard usage
- Copying diskettes
- Copying files
- Formatting diskettes
- Backing-up diskettes
- Listing the diskette directory
- Renaming files
- Erasing files
- Creating files
- Executing files
- DOS supplemental programs
- RAM disk operation

Chapter 8 contains a reference guide to the various Atari BASIC commands, operators, and functions. The following are also included:

- correct syntax for every BASIC command
- illustrative examples
- programming tips to optimize the performance of an Atari BASIC program

The final chapter contains a complete discussion of advanced memory concepts such as:

- bank switching on the 130 XE
- the binary system
- bits and bytes of data

A number of appendices are included with the **Atari XE User's Handbook**. These detail the Atari ASCII character set, BASIC reserved words, BASIC error messages, useful PEEK's and POKE's, and printer usage with the Atari XE computer.

We wish to gratefully acknowledge Pam Berliner of the Atari Corporation for her assistance in this project.

1

Introduction to the XE Series

The XE™ line represents the first products introduction by Atari since its takeover by Jack, Sam, and Leonard Trameil. At the time of this writing only the 130XE model is available, however three other XE models, the 65XE™, 65XEP™, and 65XEM™, have been announced.

One of the key features of the XE series is that they are designed to be 100% compatible with the previous XL series. Due to the similar designs around the 6502C 8-bit microprocessor, the XE series is also somewhat compatible with the original 400/800 models. The vast majority of software written for the 800XL as well as the 400/800 will run without difficulty on the XE models. If you own a package written for the 400/800 that will not run on the XE, you can generally correct the problem by purchasing a translator disk. The cost of the disk is nominal — \$9.95 plus \$2.00 shipping. It can be obtained from:

Atari Customer Relations
Atari Corporation
P.O. Box 61657
Sunnyvale, CA 94088

XE Models

As was mentioned in the introduction, four different XE models have been announced by Atari. The star and first model of the series released, the 130XE, is pictured in figures 1.1. through 1.4.

The 130XE offers the following features:

- 128K of random access memory (RAM).
- 62-key keyboard.
- Parallel interface to allow easy addition of peripherals.
- Atari BASIC programming language.
- Software cartridge slot
- Graphics and sound capability
- Expansion port
- Game controller ports
- Television jack/RF modulator*

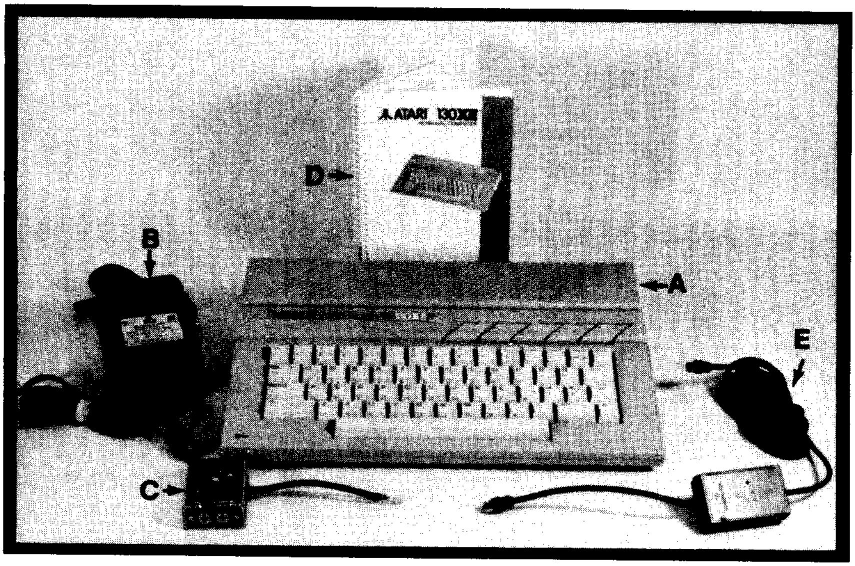
These features will be explained in greater detail in the following sections.

The member of the XE family known as the 65XE is virtually identical to the 130XE except that it contains 64K of memory rather than 128K.

The 65XEM is an enhanced version of the 65XE which will have sound generating capabilities designed around the AMIE sound chip.

The 65XEP is a portable version of the 65XE. It boasts a built in 3½″ disk drive and a built-in 5″ green monitor. At the time of this writing, neither the 65XE, XEM, nor XEP were available.

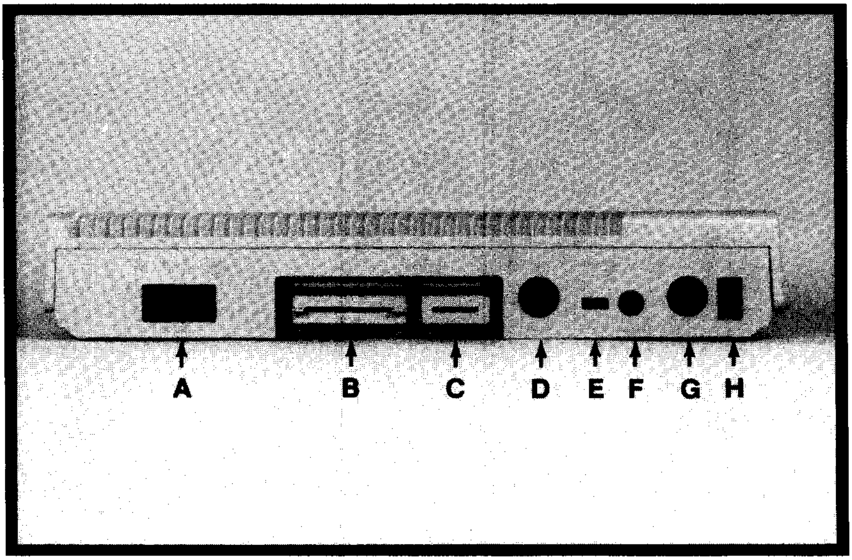
* Monitor cables must be purchased separately.



a. System unit *b.* A.C. power supply *c.* T.V. switch box
d. 130XE owners manual *e.* T.V. connect cable

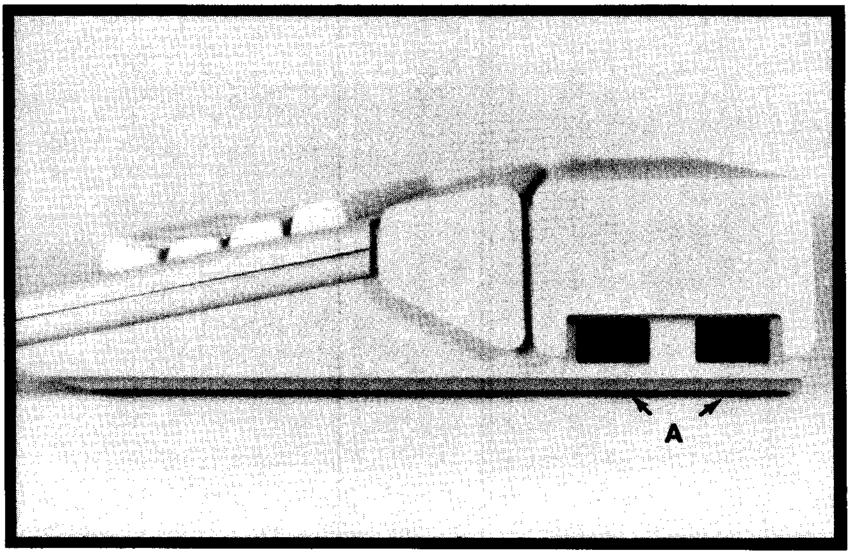
Figure 1.1. Atari 130XE (System)

Figure 1.2. Atari 130XE System Unit (top view)



a. peripheral port *b.* Cartridge slot *c.* Expansion slot *d.* Monitor jack
e. T.V. channel select *f.* T.V. jack *g.* Power jack *h.* On/Off switch

Figure 1.3. Atari 130XE System unit (rear view)



a. game controller ports

Figure 1.4. Atari 130XE System unit (side view)

Technical Data

As was mentioned in the preceding section, the system unit contains the fundamental components of the Atari. A system board or motherboard is housed within the system unit. Most of the circuitry for the Atari is located on the system board, including the following:

- 6502C microprocessor
- 64K or 128K RAM
- 24K ROM
- connectors for optional devices (POKEY)
- display hardware (GTIA, ANTIC)

ROM and 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. As previously mentioned, the Atari BASIC language interpreter is stored in ROM.

RAM stands for Random Access Memory*. Any data stored in RAM will be lost when the Atari's power is shut off. When data is loaded from the cassette unit, disk drive, or keyboard, it is stored in RAM.

DYNAMIC AND STATIC RAM

There are two different types of RAM, **dynamic RAM** and **static RAM**. Dynamic RAM can only hold the data it is storing for a few milliseconds. Therefore, any data being stored in dynamic RAM must constantly be rewritten or refreshed. This dynamic RAM refresh function must be part of ths support logic when the dynamic RAM memory is designed.

Static RAM is more expensive than dynamic RAM. However, once data has been written into static RAM, it will be retained as long as power is supplied.

The Atari computers use dynamic RAM. The custom display processor, ANTIC, has the responsibility to refresh the dynamic RAM. ANTIC's other responsibilities will be discussed later.

---

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

130XE Features

In the following sections the 130XE and its visible components will be discussed.

KEYBOARD

The 130XE's keyboard is very similar to that of an office typewriter. The individual keys are large and slightly concave to facilitate data entry. The quality of the 130XE's keyboard is suprising for a home computer. It compares well with keyboards on machines priced much higher.

The keyboard itself consists of 57 keys used to generate both alphanumeric and graphics characters. In addition, five special function keys are located on the right hand side, above the main keyboard.

PERIPHERAL PORT

The peripheral port is a serial interface which can be used to connect various devices such as disk drives, a printer, or a modem to the computer.

CARTRIDGE SLOT

This accepts Atari program cartridges. A number of Atari software applications programs are packaged in plastic cartridges containing either 8K or 16K of ROM. The cartridge slot enables the 130XE to run these programs.

EXPANSION SLOT

The expansion slot or Enhanced Cartridge Interface (ECI) allows high-speed peripheral devices such as fast floppy disk drives, hard disk drives, and custom I/O devices to be connected to the computer.

MONITOR JACK

Allows either a monochrome or color monitor to be attached to the computer. This jack uses a standard **five** DIN cable which must be purchased separately.

NOTE: Please disregard the picture on page(2) of the 130XE owners manual, as it shows the plug to be 6 DIN. It is only a **five** DIN plug! This type of plug comes with most monitors but be sure and verify this.

T.V. CHANNEL SELECT SWITCH

This is used to select which T.V. channel the computer will override. Pick the channel with the weakest incoming signal for lowest interference level.

TELEVISION JACK

This jack is used when a T.V. is to be used instead of a monitor. The computer package should contain all of the necessary adapters to hook up to a standard T.V. Check the 130XE owner's manual for installation procedures.

POWER ADAPTER PLUG

This is used to connect the computer to standard A.C. current via the Atari Model: C061982 power supply unit.

ON/OFF SWITCH

This turns the power to the computer on and off.

CONTROLLER PORTS

This connects touch tablets, numeric keypads, joysticks,and paddles controllers to the computer. Use port (1) if only one controller is used.

ATARI XE CPU

The central processing unit or **CPU** is the heart of any computer. The CPU controls all the other components of the computer.

In larger computers, the CPU and the **ALU** (arithmetic logic unit) consist of a group of IC chips each dedicated to its own task. In smaller computers, the CPU and ALU are generally combined on a single chip which is known as a **microprocessor.**

A microprocessor can be defined as a single chip which contains the logic of a central processing unit as well as any additional logic that must complement the CPU.

The Atari contains two microprocessors, ANTIC and the 6502C. ANTIC is a dedicated display processor; its main function is to relay information to the display chip (GTIA). The 6502C is a general purpose microprocessor that controls every component within the Atari. The Atari works as a team with the 6502C as the team captain. The members of this team and their responsibilities are listed in table 1.1. A rough schematic of the Atari is depicted in figure 1.5.

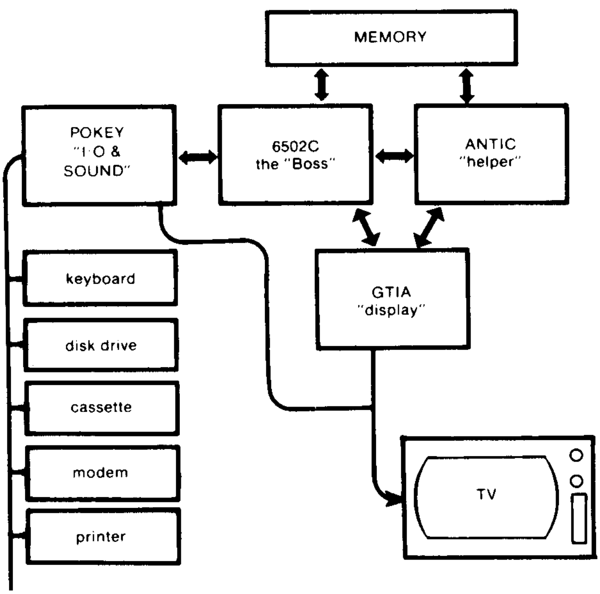


Figure 1.5. Inside the Atari

Table 1.1. Atari chips

Chip	Purpose
6502C	-general control -all numeric calculations -can handle graphics -all logical calculations
ANTIC	-display processor -relays screen information from memory to GTIA
GTIA	-convert digital information from ANTIC or 6502C into a signal that can be understood by a display device -light pen control
POKEY	-handle serial I/O to peripherals including: printer modem disk drive cassette unit -generate 4 channel music and sound effects.

Microprocessor logic is based on the **bit**. A bit is a switch that may be set to one of two states — true or false (on/off; 1/0). All information storage within a computer is based on the bit regardless of whether the information is data or commands.

Bits are often separated into groups of eight. These groups of 8 bits are known as a **byte**. A byte is required to represent a single character, (i.e. letter, number, or symbol). Collectively, 1024 bytes are known as a **kilobyte**. "K" is often used as an abbreviation for kilobyte.

Most microprocessors can address (or work directly with) 65,536 bytes (64K) at any one time. The 6502C is no exception. Even though this number appears large, a 30 page document would fill this memory area. Not to worry, 64K is quite sufficient for the majority of computer applications.

Software

Software can be defined as the set of information or programs that cause the computer to operate. Software can be divided among three general classifications:

- Operating System Software
- Language Software
- Applications Software

Each of these classes of software will be defined and discussed briefly in the context of the Atari in the following sections. Atari software can be stored on cassette tape, floppy diskettes, or cartridges.

OPERATING SYSTEM SOFTWARE

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 controlling data input/output, memory assignments, etc. The Atari operating system is stored permanently in ROM.

The operating system stored in ROM does not, however, support disk access. A program known as a disk operating system (DOS) may be loaded into the Atari to supplement its own operating system. The part of DOS that supports the disk access is known as a file management system (FMS). DOS is available in four versions — 1.0, 2.0S, 2.5 and 3. Each of these has its own file management system. DOS 2.0S and DOS 2.5 will be discussed in chapter 7, "DOS Usage".

LANGUAGE SOFTWARE

A **language** can be defined as a group of characters and/or symbols which can be combined using a set of syntax rules to represent information. Examples of languages include English, Spanish, French, as well as computer programming languages such as BASIC, LOGO, PASCAL, and COBOL. BASIC is supplied with the Atari.

Computer languages are often distinguished as being either **compiled** or **interpreted** languages. These terms refer to the way in which the program entered by the user is translated into the machine language used by the microprocessor.

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

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

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

An interpreted language consists only of source code. The source code is translated line-by-line directly into machine language instructions. The BASIC language that is standard on the Atari is an interpreted language.

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

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

BASIC programming on the Atari will be discussed in more detail in chapters 3 through 6.

APPLICATIONS SOFTWARE

Applications software can be defined as a set of instructions designed to accomplish a specific task that is of some value to the user. Examples of applications programs include games, word processing programs, spreadsheets, and database software. Generally, applications programs are stored on cassette or diskette and are transferred into RAM, where the program is available to the computer. Applications programs can also be stored in a permanent form on a ROM cartridge. This ROM cartridge can be plugged into the cartridge slot.

A large variety of applications software is available for use with the Atari. These include programs which can be used in the home such as the Home Filing Manager; programs which can be used at work such as the Bookkeeper and Visicalc; programs with educational applications such as Conversational German and Atari Speed Reading, and finally games such as Donkey Kong, Dig Dug, Defender, etc.

ATARI CARTRIDGES

As was mentioned in the preceding section, cartridges are often used to store Atari programs. An Atari ROM cartridge is pictured in figure 1.6. This cartridge consists of 16K of ROM enclosed in a plastic case. In general, cartridges have either 8Kor 16K of ROM.

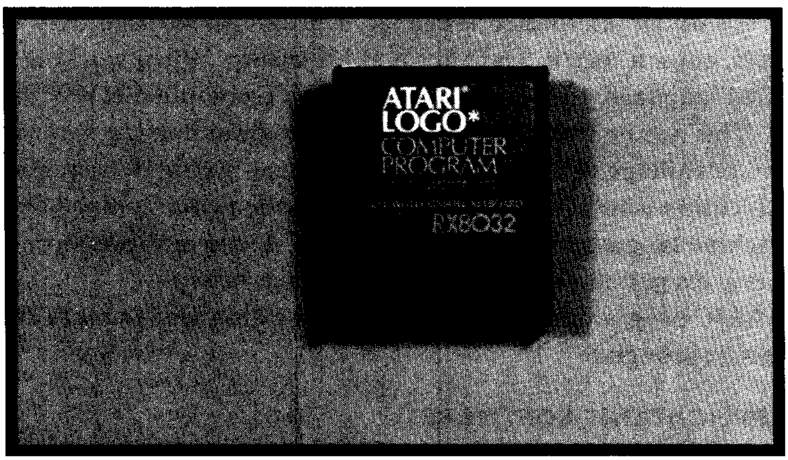


Figure 1.6. Atari cartridge

Peripheral and Add-On Devices

A **peripheral** can be defined as an auxiliary device which can be connected to a computer to perform some additional function.

A number of peripherals and add-on devices can be added to the Atari to expand it into a total computer system. These include a cassette recorder, a disk drive, printers, joysticks, a modem, and additional RAM. A number of these peripheral components will be described in the following sections.

DISK DRIVES

Atari manufacturers two disk drives that are compatible with the XE series — the 810 and 1050. The 810 can store 90K of information in that it is **single sided** and **single density**. The 1050 can store 90K in its single density mode, while storing 130K in **dual density**. Both drives are **soft sectored.**

The disk drive is one of the more important parts of a computer system. Strong consideration should be given to the purchasing of a disk drive because it allows the storage of relatively large amounts of data and also offers relatively fast access to that data.

Unlike RAM storage, when information is stored on a disk, the information is not lost when the computer is turned off. In other words, disks offer a permanent means of storing data.

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 lies in the means by which that data can be accessed.

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 device.

FLOPPY DISKETTES

Disk drives store data on floppy **diskettes.** 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 figure 1.7.

The diskette is allowed to rotate within the protective cover. 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 cover provides an area where the head can read from or write to the diskette surface.

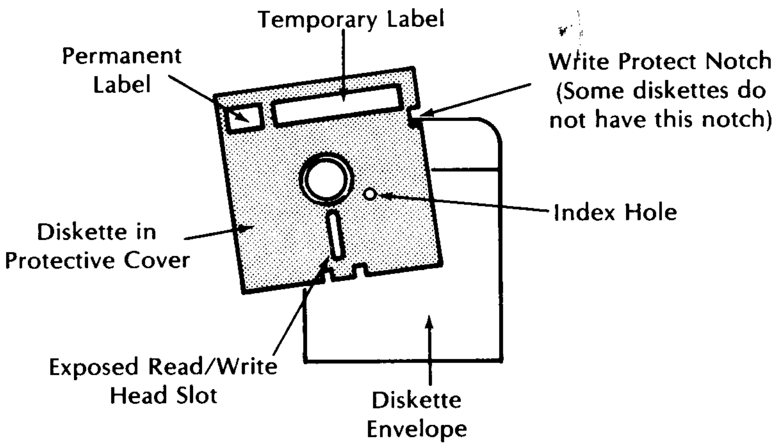


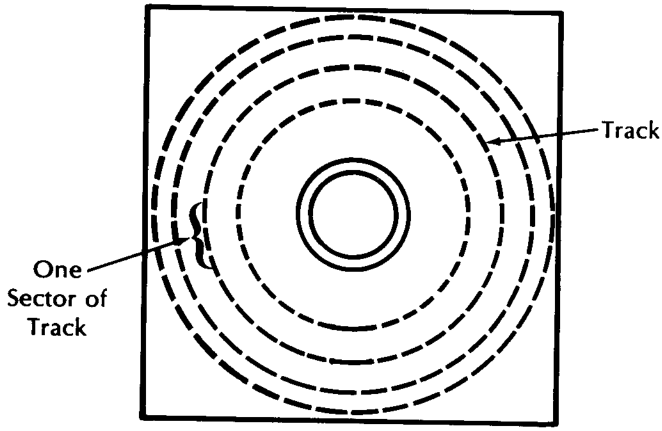
Figure 1.7. 5¼ inch floppy diskette

TRACKS AND SECTORS

To facilitate the processs 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 figure 1.8. Both single and dual density divide one side of the disk into 40 tracks. The other side is not used (single-sided).

To further reduce the time necessary to search for a particular data item, single density divides each track into 18 sectors, also shown in figure 1.8. Dual density divides each track into 26 sectors.

Each individual sectors 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.



Single density:

$$\frac{\text{40 tracks}}{\text{1 disk}} \cdot \frac{\text{18 sectors}}{\text{1 track}} \cdot \frac{\text{128 bytes}}{\text{1 sector}} = \frac{\text{90K}}{\text{disk}}$$

Dual density

$$\frac{\text{40 tracks}}{\text{1 disk}} \cdot \frac{\text{26 sectors}}{\text{1 track}} \cdot \frac{\text{128 bytes}}{\text{1 sector}} = \frac{\text{130K}}{\text{disk}}$$

Figure 1.8. Tracks and sectors

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 figure 1.7. 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 figure 1.7 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 figure 1.9.

Soft sectored diskettes have only one index hole as shown in figure 1.10. 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 drives use soft sectored diskettes.

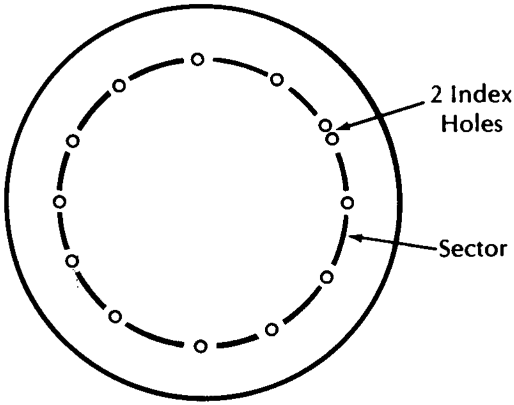


Figure 1.9. Hard sector diskette

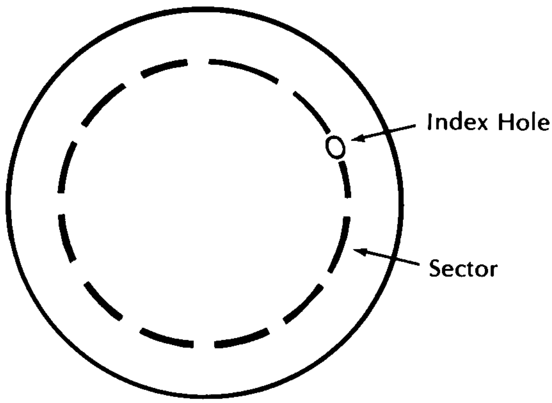


Figure 1.10. Soft sectored diskette

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 5¼ inch diskettes this notch is known as a write-enable notch.

Information cannot be written onto a 5¼ inch diskette unless the write-enable notch has been left uncovered.

Some 5¼ inch diskettes (especially system diskettes) may be permanently write protected if their protective envelopes 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 figure 1.11.

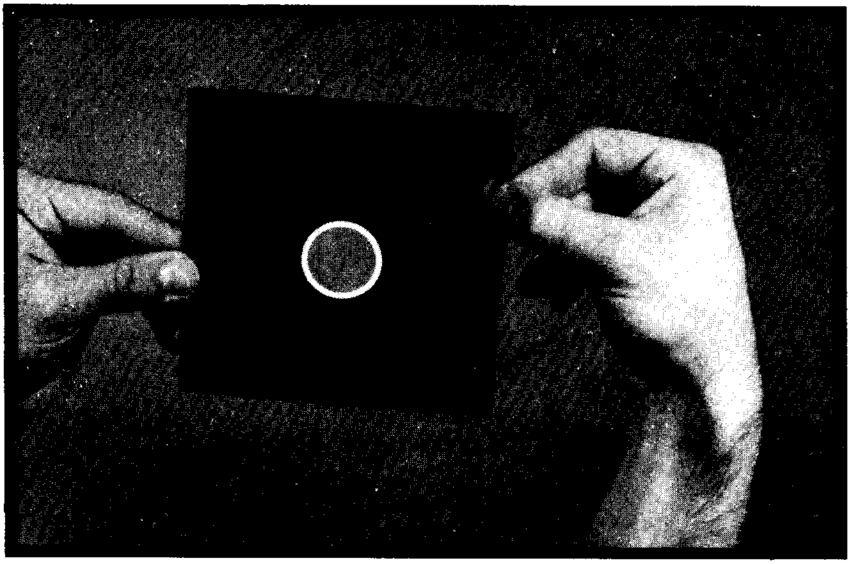


Figure 1.11. Write protecting a 5¼ inch diskette

DISK DRIVE OPERATION

1050 disk drive operation is a relatively simple mater. When there is no diskette in the disk drive, the disk slot handle should be in the horizontal or open position (see figure 1.12).

When inserting a diskette, the diskette's label should be facing up. The edge of the diskette closest to the oval-shaped opening in the cover should be inserted into the drive (see figure 1.13).

Slide the diskette into the diskette slot. Once the diskette has been fully inserted, rotate the diskette slot handle to the vertical or closed position. To remove a diskette from the drive, merely reverse this procedure.

810 disk drive operation is similar to that of the 1050; however the 810 has a door which covers the disk slot. This door may be opened by pressing the button directly beneath the door, (see figure 1.14). The disk should be inserted into the slot, then the door should be closed.

In general, a disk drive has a small red lamp on its front cover. This lamp will light whenever data is being read from or written to a diskette. Do not remove a diskette when this lamp is on.

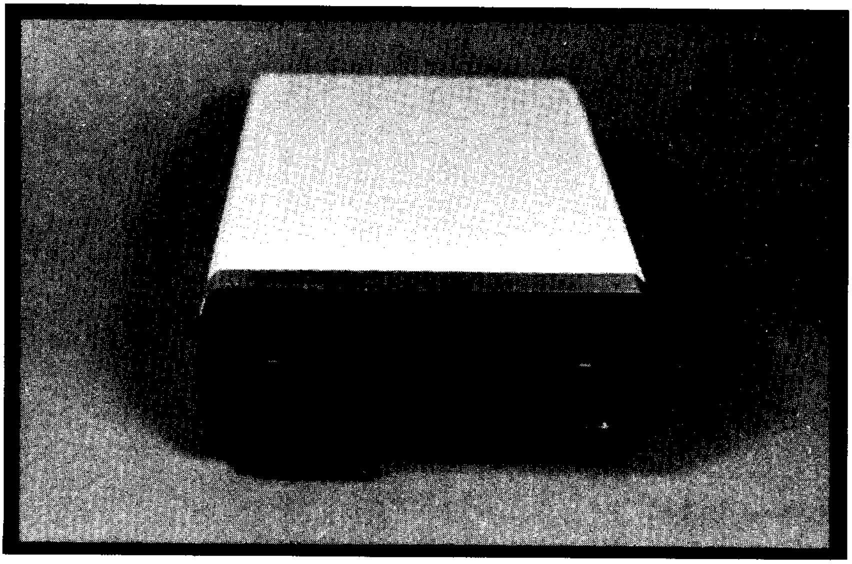


Figure 1.12. Diskette slot handle in open position (1050)

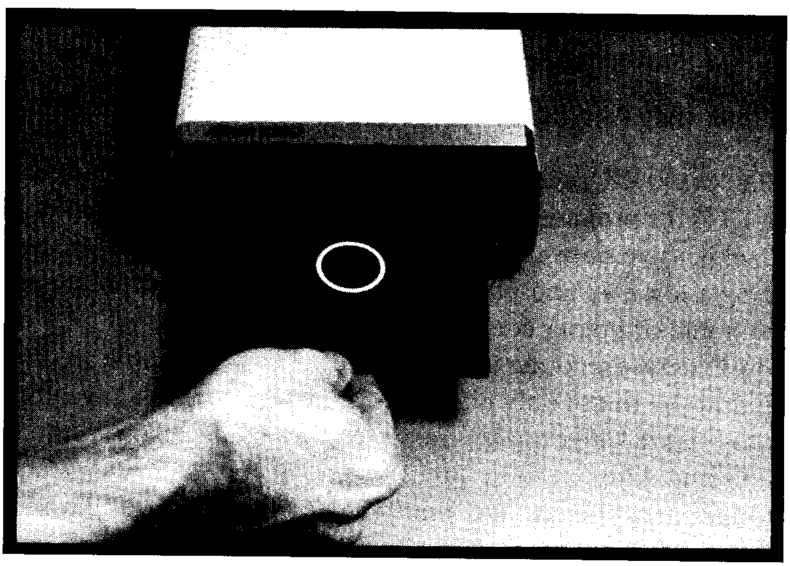


Figure 1.13. Inserting a diskette into the Atari drive

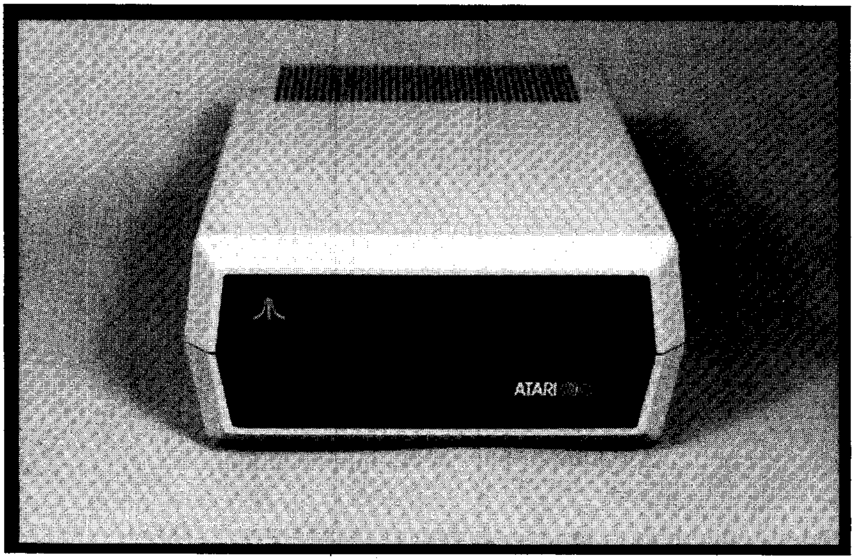


Figure 1.14. The 810 disk drive with door closed

ATARI 1010 PROGRAM RECORDER

The Atari XE computers can utilize the 1010 Program recorder (figure 1.15) for program and data storage. The Program recorder is the most inexpensive data storage device available for a home computer, providing a low cost and reliable means of data storage for the budget-minded home computer consumer.

The Program recorder uses standard cassette tapes to store data. It is a good practice to use only high quality cassette tapes to save programs and data. Using lesser quality cassette tapes could result in the loss of programs and data.

Besides data storage, the Atari 1010 can also store audio information. This technique is used in the following computerized language courses: ATARI Conversational French, German, Spanish, and Italian.

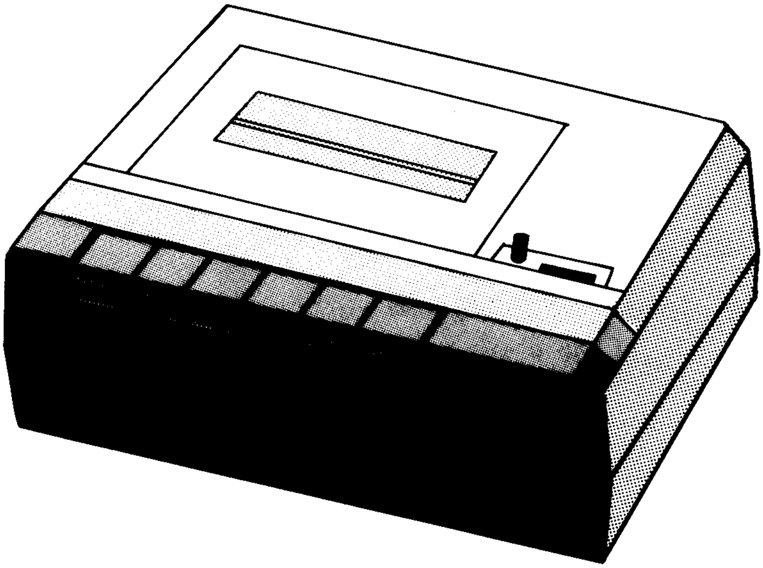


Figure 1.15. Atari 1010 Program recorder

Printers

Printers used with personal computers can be classified among two major types — **dot matrix** printers and **daisy wheel** printers. Dot matrix printers output characters on paper as a group of dots. Dot matrix printers output data at speeds ranging from 35 to 350 characters per second (or 400 to 4000 words per minute).

Daisy wheel printers output characters that appear much like those output by a typewriter. The only difference is that a daisy wheel printer uses a round printing element which contains the standard character set. The wheel spins to the correct position each time a character is to be printed. Daisy wheel printers are generally more expensive as well as slower than dot matrix printers. However, the quality of the characters output by the daisy wheel printers is higher that those output by dot matrix printers.

Printers used with personal computers are generally either serial or parallel devices. In **serial** communications, data is transferred one bit at a time from the source device to the receiving device. In **parallel** communications, data is transferred eight bits (or one byte) at a time.

ATARI 1020 COLOR PRINTER

The 1020 Color Printer specializes in printing four color graphics and text. This printer draws with 4 pens (red, blue, green, and black). Because the 1020 does not use a ribbon, as do most printers, it is suited for only intermittent use.

ATARI 1025 80-COLUMN PRINTER

Atari 1025 80-column printer may be also be used with an XE series computer. The 1025 is a serial dot-matrix printer with a throughput of 40 characters per second (40 c.p.s.). At this speed, it is the quickest of the Atari printers; it is also the most durable.

ATARI 1027 LETTER QUALITY PRINTER

Like the 1025, the Atari 1027 (figure 1.16) letter quality printer is a serial device capable of an 80-column output. The 1027 runs at half the speed of the 1025, (20 c.p.s.). However, the 1027 is **letter quality**. This printer is similar in operation to a daisy wheel printer — producing typewriter quality text.

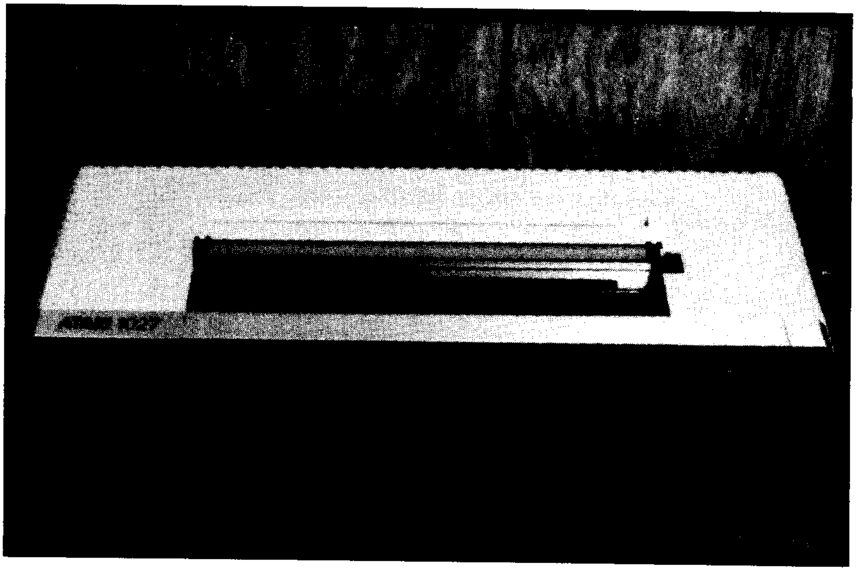


Figure 1.16. Atari 1027 printer

ATARI 850 INTERFACE MODULE

The interface module (figure 1.17) is a device that converts the serial output of the XE computers into a parallel output. This device will allow the connection of many parallel printers not manufactured by Atari to the XE series.

The 850 also contains 4 serial ports. These ports are RS-232* compatible. There are a plethora of peripherals available that use the RS-232 standard, including:

- Printers
- Modems
- Voice Synthesizers

Specifically, the Atari 830 acoustic modem attaches directly to the 850 interface module.

* RS-232 is the industry standard for serial communications.

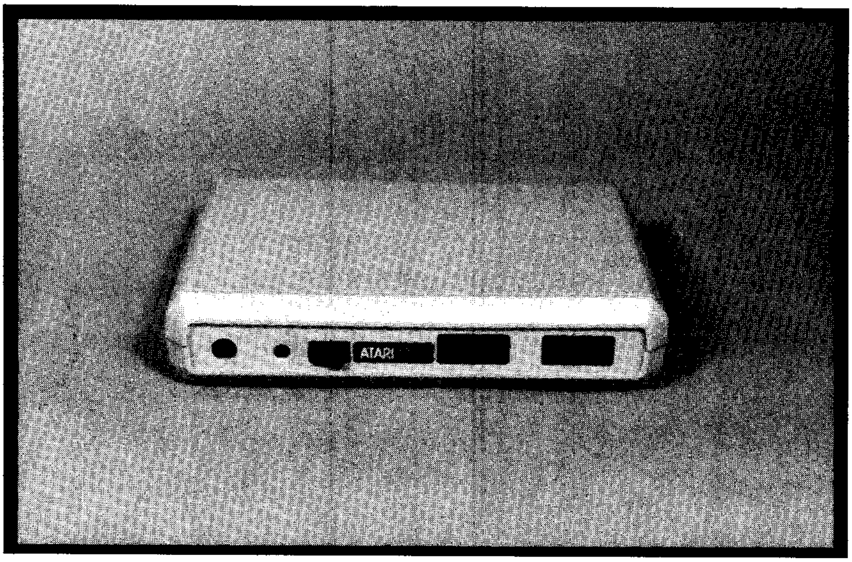


Figure 1.17. 850 interface module

ATARI MODEMS

Atari manufactures two models of modems, the 830 acoustic modem (figure 1.18) and the 835 direct-connect modem. A **modem** is a device that prepares data for transmission. Modems are generally used with computers to encode data into a series of tones that can be transmitted over telephone lines. Modems can also be used to receive and decode this data. A typical modem link is depicted in figure 1.19. The Atari 130XE on the right has connected itself, via the telephone lines, to a mainframe computer.

An **acoustic** modem is connected to the handset of the telephone, while a direct-connect modem is connected directly to the phone line. A direct-connect modem is generally more convenient because it has the capability to dial a phone number in addition to just transmitting information.

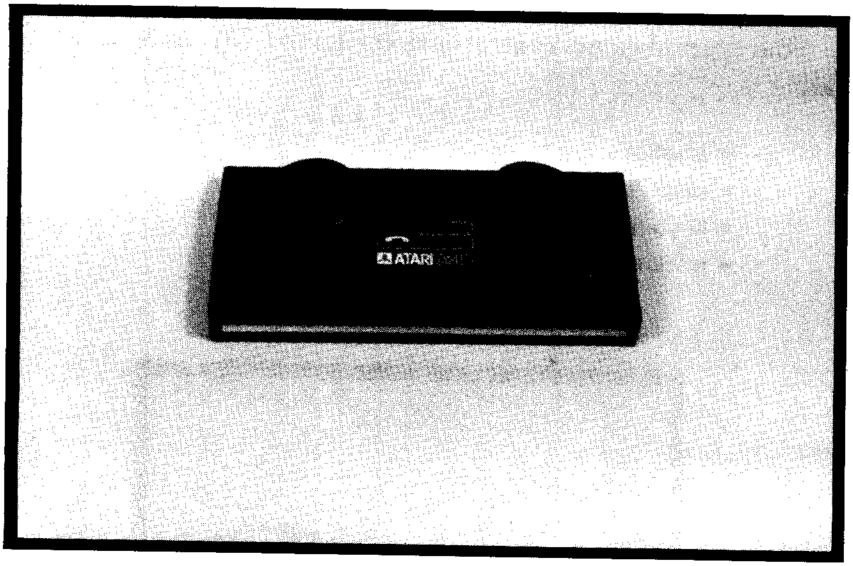


Figure 1.18. 830 acoustic modem

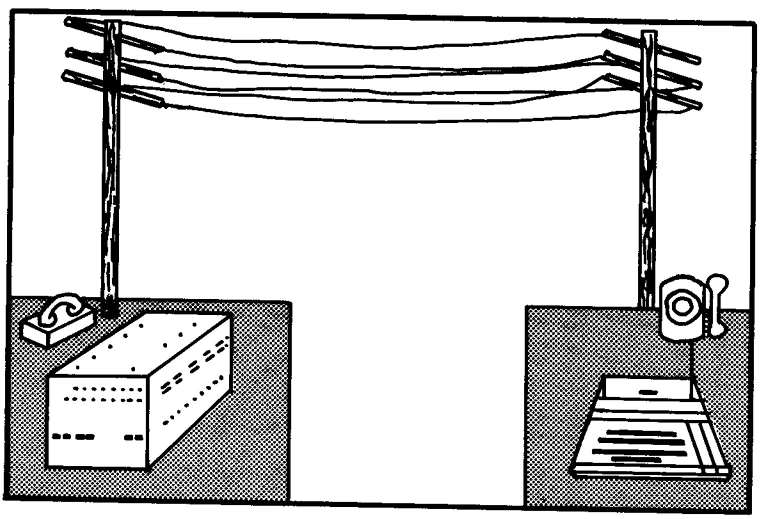


Figure 1.19. Computer/modem link

GAME CONTROLLERS

Four types of games control devices (figure 1.20) can be used with the Atari:

- joysticks
- paddles
- keyboard controllers
- track balls

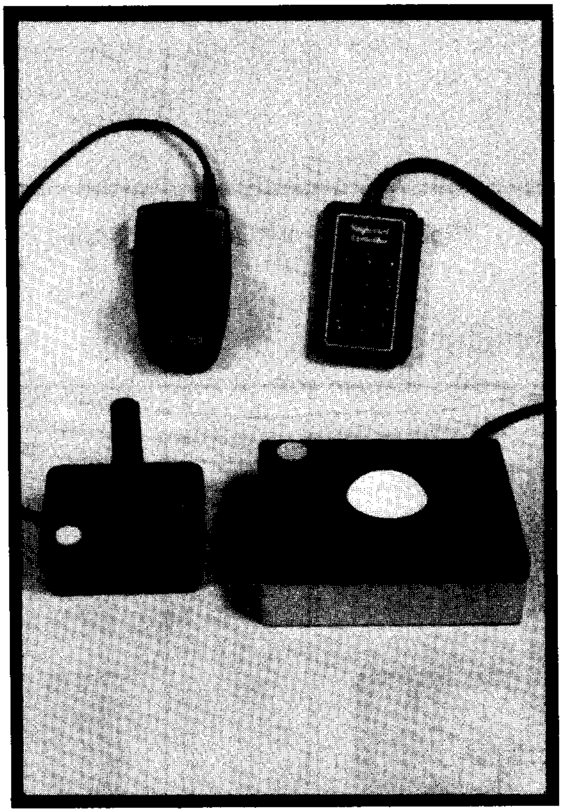


Figure 1.20. Game controllers

2

Installation, Operation, & Keyboard Usage

Introduction

The steps necessary for setting up an Atari will be explained in this chapter. Atari has simplified the installation procedure to the point where almost anyone can set up the unit. This involves unpacking the various system components and attaching the necessary cables.

This chapter will also explain Atari keyboard usage. The keys on the 130XE 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.

Installation

First of all, when unpacking the Atari XE, save the carton and packing material. These should be used if the XE is to be moved or stored in the future.

The Atari is easy to install. The Atari, television set or monitor, and any peripherals should first be positioned so that they may be easily accessed. At least two AC electrical outlets will be needed — one each for the Atari, the television or monitor, and any peripherals.

Locate the Atari's ON/OFF switch on the back left side of the console and be certain that it is positioned to OFF. Next, plug the power supply unit's cord into an AC electrical outlet. Plug the other end into the Atari's POWER IN (DC) socket. (*see* figure 2.1)

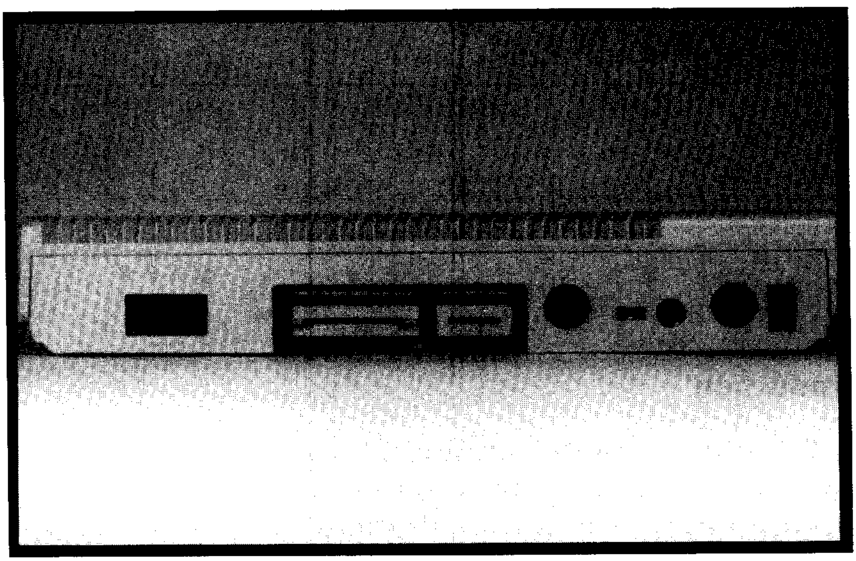


Figure 2.1. Rear of Atari 130XE

The system unit must be attached to either a television set or a monitor. If a television set is used, refer to the section entitled "Attaching a Television Set". If a monitor is to be used, refer to the section entitled "Attaching a Monitor".

ATTACHING A TELEVISION SET

To use a television set as a display device, first connect the video cable to the RCA jack on the rear of the Atari. This jack is labeled TELEVISION. The video cable, shown in figure 2.2, contains a small box near one end. This end should be connected to the computer.

The next step is to install the TV Switch Box on the television set (see figure 2.3). The TV Switch Box has been designed so that it can be permanently installed on your television, as it allows regular TV reception as well as video output for the Atari. The Switch Box has an adhesive backing that can be used to attach it to the back of your television.

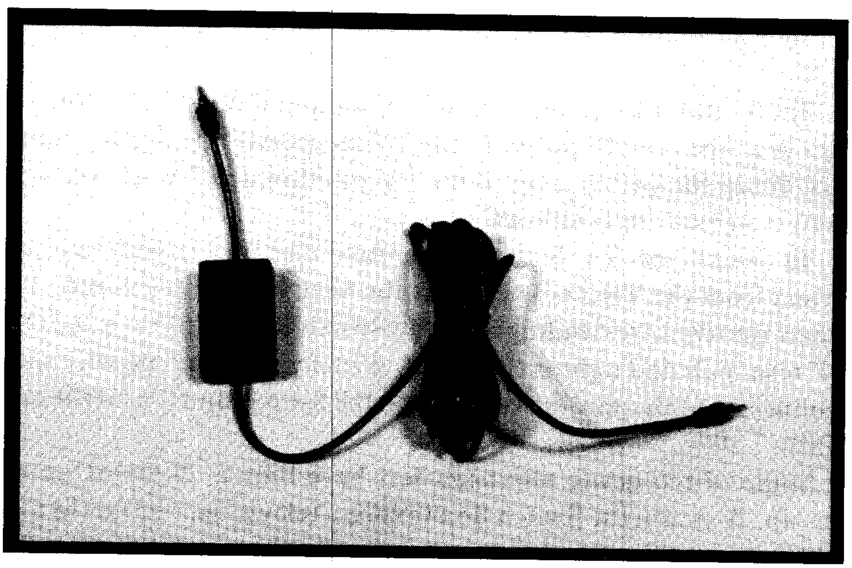


Figure 2.2. Video cable

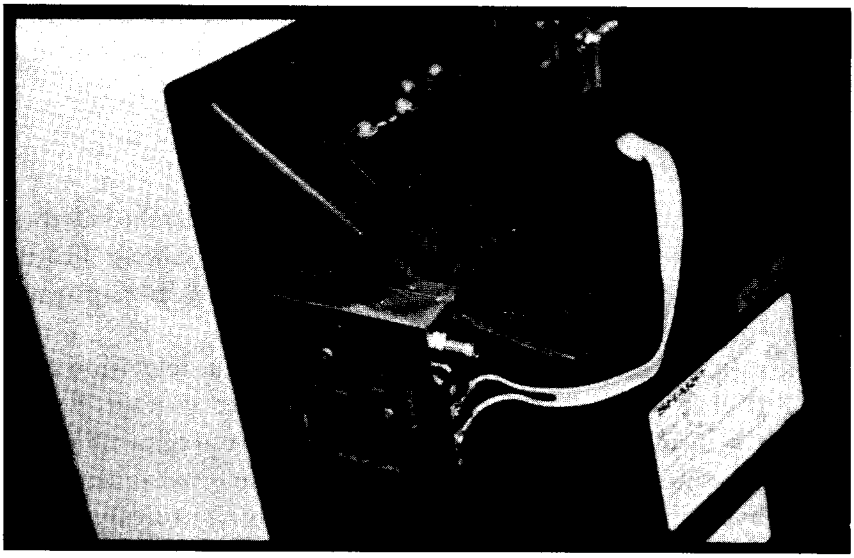


Figure 2.3. TV Switch Box

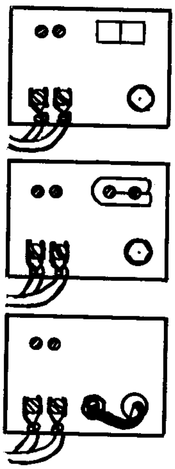
The Switch Box contains a switch marked GAME/TV. When this switch is at the GAME position, the TV set recevies its signals from the Atari. When the switch is set to the TV position, the TV set receives its signals from your television antenna.

To install the TV Switch Box, first disconnect your television antenna from the VHF connector at the back of your television. Then, connect the two wires leading from the Switch Box to the twin 300 OHM VHF terminals and tighten the screws. Finally, the antenna wire should be connected to the appropriate plug on the Switch Box (75 OHM or 300 OHM).

Some of the newer television sets have only a 75 OHM antenna hook-up. To attach the Switch Box to such a television, a 75 OHM to 300 OHM converter must be used between the television and the Switch Box. If your television has a cable hook-up, treat the wire from the cable box to the TV as the antenna wire in the previous discussion.

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 figure 2.4. 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.



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

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

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

Figure 2.4. 300 OHM conversion

Next, connect the Video Cable to the Switch Box. Be certain that the slide in the center of the Switch Box is set to GAME and then turn on the TV set.

Your TV set should be turned to VHF channel 2 or 3. The channel chosen should correspond to the setting of the Atari's 2-CHAN-3 switch. A channel not used by a local television station should be selected. If both stations are used in your area, select whichever channel has the weakest reception.

ATTACHING A MONITOR AND/OR AUDIO SYSTEM

The Atari 130XE can also be connected to a monitor for visual output and/or to a stereo for audio output.

The connection is accomplished through the MONITOR jack on the rear panel of the 130XE. A standard 5-pin DIN audio cable can be used to make the physical connection. This cable is not supplied with the 130XE, but can be obtained from most electronic or audio stores. Appendix D contains the pinouts of the MONITOR jack.

INSTALLING AN ATARI DISK DRIVE

As mentioned in chapter 1, Atari markets two disk drives — the 810 and the 1050. The installation procedures for these are virtually identical.

Before installing the disk drive, be certain that the power switches on both the drive and the computer are off. The first step is to select a drive number. Any drive may be assigned any number (1-4); however, drive numbers may not duplicate and one of the disk drives must be assigned drive number 1.

The drive number is set with the two DRIVE SELECT switches (or drive code switches). These switches on the rear panel of the drive may be set with a pencil or other slender object. Figure 2.5 illustrates possible switch positions and the corresponding drive number assignment.

Next, plug one end of the data cord to the PERIPHERAL connector on the rear panel of the Atari console. The other end of the data cord should be inserted into either of the I/O connectors on the rear of the drive. Additional peripherals can be attached via the unused I/O connector.

Finally, the power supply should be connected. Plug one end of the power supply into a household outlet, and the other end into the POWER IN connector on the rear of the disk drive.

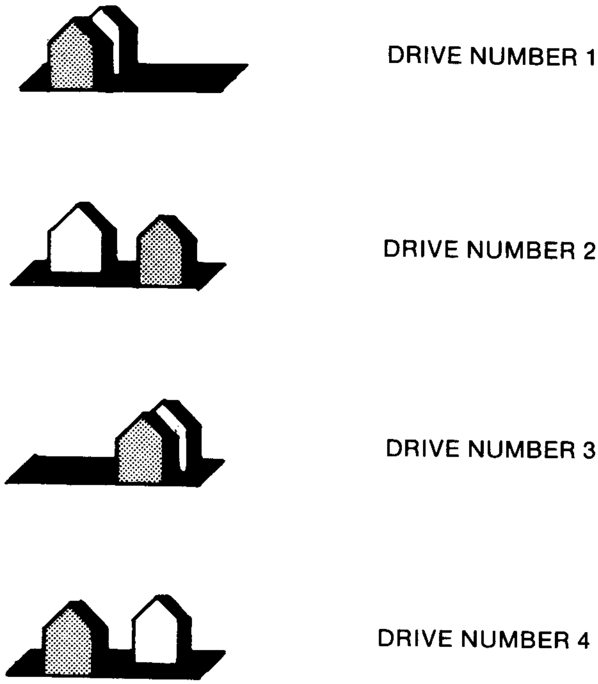


Figure 2.5. DRIVE SELECT switches

DAISY-CHAINING

The majority of the peripherals available for the Atari are connected to the serial I/O chain. This chain is started at the PERIPHERAL jack on the rear of the computer and then connects, in turn, to each peripheral. Figure 2.6 helps to clarify this concept.

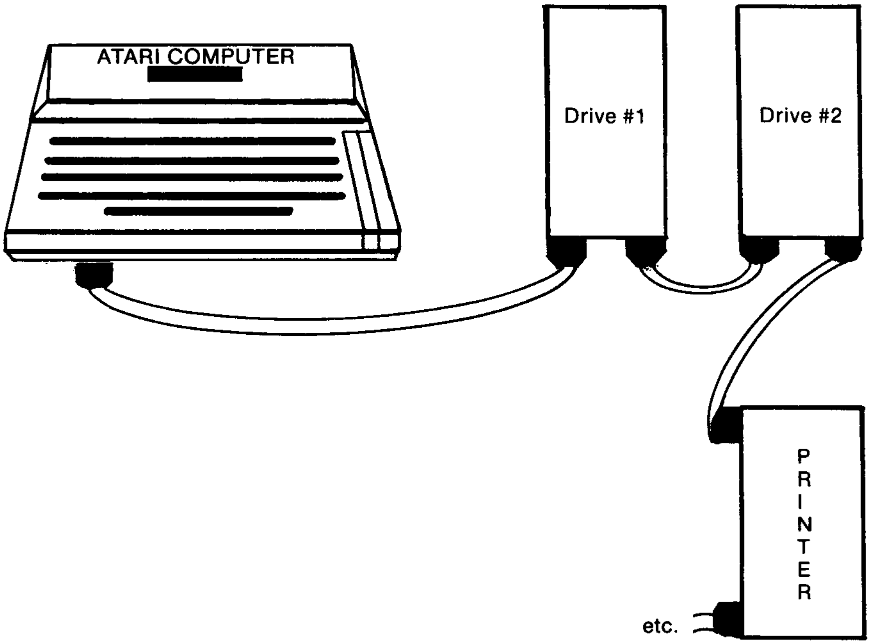


Figure 2.6. Daisy chaining

INSTALLING THE ATARI 1010 PROGRAM RECORDER

The first step in installing the Atari 1010 is to plug the data cord into one of the I/O connectors on its rear panel. Next, attach the other end of the data cord into the daisy chain of peripherals. Finally, the power cord should be attached.

Physically, more than one cassette unit may be attached to the peripheral chain. However, the operating system will be unable to distinguish them. Backups of commercial cassette software may be made using a direct connection between two 1010's. This method by-passes the computer entirely, allowing virtually any program tape to be duplicated.

INSTALLING THE ATARI 825 OR OTHER PARALLEL PRINTER

The Atari 850 interface module is required to install the Atari 825 or other parallel printer. The interface module converts serial data from the computer into the parallel data used by these printers. Figure 2.7 illustrates this connection. Usually, the printer cable must be purchased separately.

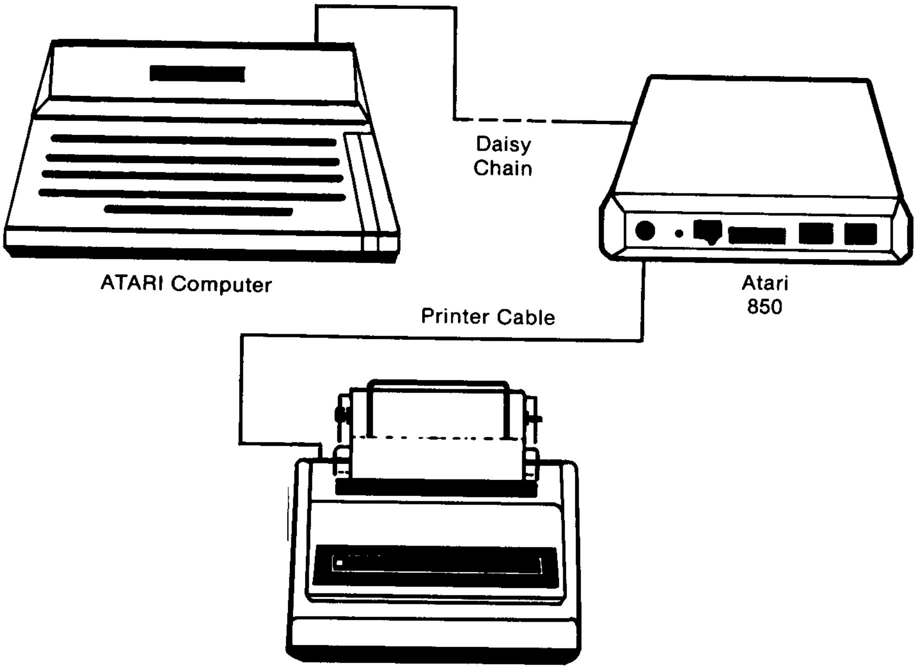


Figure 2.7. Parallel printer connection

INSTALLING THE ATARI 830 MODEM OR OTHER RS232C DEVICE

The Atari 850 interface module may also be used to connect up to four RS232 compatible devices. RS232, the industry standard for serial communication, can be used to connect devices such as modems, single-board computers, data acquistion hardware, and some printers, to the Atari. Again, a cable must usually be purchased separately.

Operation

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 box connected to the television set should be placed in the game position.
2. If you are using an Atari 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. Be certain that the correct ROM cartridge has been installed, and then turn on the Atari computer console.
5. Turn on the printer when you wish to use it. Remember, if you are using a parallel 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.

Keyboard Usage

The Atari keyboard contains most of the same keys arranged in the order of a regular typewriter keyboard (see figure 2.8). The Atari keyboard also contains several additional keys not found on the typewriter keyboard. Two of these, ESC and CONTROL, are located on the left side of the keyboard. Three other keys, BREAK, CAPS, and are located on the right side of the keyboard. Also, in the upper right-hand side of the keyboard are five special function keys. Finally, some of the standard typewriter keys contain special words or special symbols.

Every key except SHIFT, CONTROL, BREAK, and RESET has a built-in auto-repeat feature. Auto-repeat means that when a key is held down, that character will be repeatedly output until the key is released. For example, if the A key is pressed, a single A will be displayed on the screen. After a second or two, the A will be repeated on the display as long as the A key is depressed.

Experimentation is encouraged as the following paragraphs are read. Do not worry about damaging the computer. Any error situation caused by keyboard entries can be corrected by merely turning the Atari off, then on again.

Figure 2.8. Atari 130XE keyboard

RESET

The RESET key is located at the top and at the far right of the keyboard. When the RESET key is pressed, all computer operations stop, and the Atari is restarted. In other words, control is generally returned to the operator.

Be careful not to press RESET accidentally. Doing so can cause the loss of data — especially if the disk drive is in use when RESET is pressed. Generally, if the rest of the keyboard will not respond, it is appropriate to use the RESET key.

OPTION, SELECT, START

The functions of these three keys may be programmed for each specific application. Generally, they are used to choose options within commercial programs.

The OPTION key and START key have special meanings during power-up. The OPTION key may be used to disable Atari BASIC. Depressing the OPTION key during power-up will replace the 8K BASIC ROM with 8K RAM. The START key is used to load machine language cassette programs. Depressing START during power-up causes the television speaker to emit a single tone that signals that the Atari is ready to accept a cassette program. Pressing the RETURN key, here, will boot the program.

HELP

The HELP key has been added to the XE series of computers to allow programs to have an on-line help feature. This feature is mainly used in commercial software programs.

RETURN

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

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 the 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.

SHIFT

Upon start-up, the keys for the letters (A-Z) always produce uppercase 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 that are affected by the position of the SHIFT key include those with more than one character displayed on their top. The character nearest the user is output when the SHIFT key is not pressed. The character nearest the upper-right corner of the key is output when the SHIFT key is pressed.

In this book, a key produced in the SHIFT mode will be denoted by the word SHIFT followed by the character produced without the SHIFT key. For instance, SHIFT-8 would denote the symbol @. Appendix B lists the characters produced in the SHIFT mode.

CONTROL

The CONTROL key is used in combination with another key much as the SHIFT key is. CONTROL 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 CONTROL. For example, CONTROL-C designates holding the CONTROL key while pressing the C key.

CONTROL is used with the letter keys to output the graphics characters, and with other keys to instruct the computer to undertake a particular function. For example, CONTROL- = causes the cursor to move one row down. The CONTROL key combinations are listed in appendix B.

CAPS

As mentioned earlier, upon start-up, the keys for the letters (A-Z) always produce uppercase or capital letters, regardless of whether the SHIFT key is depressed or released. The CAPS key allows both capital and lowercase letters to be output.

To output both capitals and lowercase letters, press the CAPS key once. Now, when the SHIFT key is released, lowercase letters will be output; however, when the SHIFT key is depressed, uppercase will still be output. Pressing the CAPS key a second time will return the Atari to the all uppercase mode.

At any time, pressing the SHIFT-CAPS key combination results in the all uppercase mode. The keyboard can be placed in the graphics characters mode by pressing the CONTROL-CAPS key combination.

The  key is used to toggle the keyboard between the normal and reverse video modes. In the reverse (inverse) video mode, the background and foreground colors are exchanged when displaying subsequent characters.

BACK SPACE

The Bk Sp key moves the cursor one position to the left each time it is pressed. The character beneath the cursor will be erased when Bk Sp is pressed. If the cursor is at the left edge of the screen and Bk Sp is pressed, the cursor will not move.

CLEAR

Either the SHIFT-< or CONTROL-< 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 corner of the screen.

DELETE

Individual characters can be deleted from the line in which the cursor resides. CONTROL-DELETE causes the character at the cursor position to be deleted. The characters to the right of the cursor will be moved one space to the left to fill the void.

SHIFT-DELETE causes the line that the cursor is currently in to be erased from the screen. Then the lines beneath that line will be shifted upward in the display.

INSERT

CONTROL-> will insert blank spaces at the cursor position. The characters to the right of the cursor will be moved one position to the right.

SHIFT-> results in a blank line being inserted at the cursor position. The remainder of the display below the current line is moved down by one line.

TAB

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 desired stop. Pressing CONTROL-TAB clears the tab stop at the cursor's current position.

ESC

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 will not clear the screen as CONTROL-< usually does:

ESC CONTROL-<

ARROW KEYS

The arrow keys are generally used to move the cursor on the screen, so that the keyboard entries can be corrected where necessary. The arrow keys are generated using the following CONTROL key combinations:

↑	CONTROL- —
↓	CONTROL- =
→	CONTROL- *
←	CONTROL- +

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 the 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 keys move the cursor up and 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 of the screen, down arrow places it at the screen's top.

3

Introduction to the Atari BASIC

Introduction

In this chapter, the operating details necessary to begin using Atari BASIC will be provided. These include start-up, program entry, statement structure, program editing, program saving, and program listing. In addition, the fundamental concepts necessary to master Atari BASIC will be examined. Especially, the various **data** types used in Atari BASIC as well as the **operations** that can be performed on that data will be discussed.

Getting Started with Atari BASIC

Atari BASIC is a high-level language that must be interpreted into the microprocessor's native language. This is accomplished with a program known as an **interpreter.** The correct activation procedure for the BASIC interpreter is determined by the model of computer being used. For example, the 1200XL requires that the BASIC cartridge be inserted; whereas BASIC is built into the 130XE.

The correct BASIC start-up procedure also depends on whether or not the system contains a disk drive. Both methods will be detailed in the following sections.

START-UP WITHOUT A DISK DRIVE

This section describes how to start-up BASIC without a disk drive. Therefore, if the system includes a disk drive, skip this section and proceed to the next.

If a cartridge is presently inserted in the cartridge slot on the back of the unit, it should be removed at this time. (The cartridge slot is located on the back in the center of the computer.)

The computer itself should now be powered up. If the Atari is already activated, it should be turned off, then reactivated. A clear blue display will appear while a series of initialization procedures are performed. About 4 seconds later, the "READY" prompt and the cursor will appear at the top of the screen (see figure 3.1).

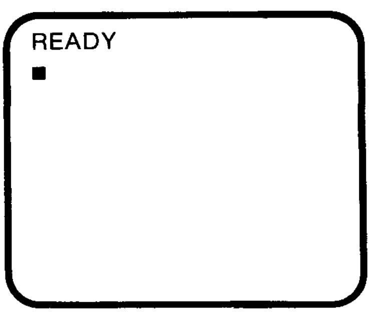


Figure 3.1. Start-up display

START-UP WITH A DISK DRIVE

The Atari BASIC interpreter, by itself, does not support disk access. However, disk access is supported by the interpreter if the disk operating system has been loaded into memory.

To load DOS into the computer, the disk drive must first be powered-up. Also, a system diskette containing a copy of DOS should be inserted into the disk drive. Remember, never use the original copy of DOS. Make a copy, and then store the original in a safe place. Always use a copy for everyday use. Chapter 7, "DOS Usage", explains how to make back-ups.

If a cartridge is presently inserted in the cartridge slot on the top of the unit, it should be removed at this time. (The cartridge slot is located on the back in the center of the computer.)

The computer itself should now be powered-up. If the Atari is already activated, it should be turned off, then reactivated. A clear blue display will appear while DOS is loaded into memory. The television speaker will emit a series of beeps at this time. After the loading of DOS is complete, about 4 seconds of initialization procedures will be performed. These will also be audible, producing a sputtering sound. Finally, the "READY" prompt and the cursor will appear at the top of the screen.

IMMEDIATE AND PROGRAM MODES

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

```
PRINT "Walter A. Haupt"
```

the following would be displayed on the video screen:

Walter A. Haupt

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

Figure 3.2 contains an example of the entry of a program in the program mode and its execution. Notice that in the program mode, each BASIC program line must be preceded with a line number. Line numbers will be discussed in more detail later in this chapter.

```
10 PRINT "Walter A. Haupt"
20 PRINT "24270 Glenbrook"
30 PRINT "Euclid, OHIO 44112"
40 END
RUN
Walter A. Haupt
24270 Glenbrook
Euclid, OHIO 44112

READY
```

Figure 3.2. Program mode entry and execution

COMMAND AND STATEMENT STRUCTURE

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

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

In Atari BASIC, keywords must be entered in uppercase letters. The Atari will not recognize an entry as a program line unless its keyword is capitalized. An error will result if a lowercase entry is made.

A BASIC command or statement generally includes one or more **arguments** or **parameters** following the keyword. In our example, "Walter A. Haupt" is the PRINT statement parameter.

```
PRINT "Walter A. Haupt"
```

ENTERING A PROGRAM

In the preceding section, the fundamentals of entering and running an Atari BASIC program have been touched upon. In this section, that discussion will be expanded upon by using the example in figure 3.3.

BASIC programs are entered as **program lines.** Any text preceded with a number (**line number**) and ended by pressing the RETURN key will be regarded as a program line. The maximum number of characters that may be included in any one line is 114. Line numbers must be integers in the range 0 to 32767. If a line exceeds its character limit, only the first 114 characters will be remembered. A bell will sound after the 107th character is typed as a reminder that the limit is being approached. If a line number is not valid, an error condition results.

Note that in the first 7 lines of figure 3.3, a program was entered in the command mode and run in the execute mode. After the answer, 5, had been displayed, the "READY" prompt appeared.

At this point, the original program will be stored in memory, and can be added to or changed. That is what was done in line number 150 of figure 3.3. An additional statement was inserted between statements 100 and 200 in the program being stored in memory. This revised program can be executed by again entering RUN.

The computer memory can only hold one program at a time. The NEW command is used to erase the program in memory so as to allow a new program to be entered. Note the use of NEW in figure 3.3.

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

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

```
NEW

READY
100 PRINT 5
200 END
RUN
5

READY
150 PRINT -5
RUN
5
-5

READY
NEW

READY
100 PRINT 50
200 END
RUN
50

READY
```

Figure 3.3. Entering and running a program

It is recommended that consecutive line numbers (10, 11, 12, 13, etc.) not be used in programs. By using numbers that are a fixed distance apart (100, 110, 120, 130, etc.), additional lines can be inserted between existing lines without renumbering the lines.

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

If two lines are entered with the same line number, the original line will be erased, then replaced with the new line. This feature allows the user to replace an entire line by merely entering a new line with the same line number.

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

To delete a line in an existing program, merely enter the line number of the line to be deleted followed by RETURN. Of course, an entire program may be deleted with the NEW command.

ERROR MESSAGES

When a statement of incorrect format has been entered, an error message will be displayed. Only syntax errors are detected when a program is being entered. If an incorrect line is typed and RETURN is pressed, the Atari will print an error message, followed by the line just entered. The location of the error within the line will be identified with an inverse video character. In the following example, the operator's entry is signified with bold face, while the computer's response appears in normal type:

PRINT 5G
ERROR - PRINT 5 G

location of error

If a problem develops while a program is being executed, an error message will be displayed. An error that occurs during the execution of a program will generate an error message that includes a numeric description of the problem as well as the line number of the statement that caused the problem. The numeric description is a code that indicates the nature of the error. The various error codes and their corresponding descriptions are listed in appendix A.

When an error occurs in a program, an error message will be displayed and the execution of the program will halt. Here, program execution may be resumed by using the CONT command.

LISTING A PROGRAM

LIST is used to display the program stored in memory on the screen. This display is often referred to as a **program listing.** An example of the use of LIST is given in figure 3.4.

When the LIST command is executed, the program in the computer's memory will be displayed on the screen. Each line of the program appears initially at the bottom of the display. In order for each subsequent line of the program to appear on the last line of the display, each line of the display must be moved one line toward the top. As a result, if a program occupies more than 24 display lines, the first lines of the program will be moved off the top of the display in order to accommodate the last lines. This process is called **scrolling.**

When a lengthy program is listed on the display, the information may pass by too quickly to be usable. As a result, it is often necessary to temporarily halt the listing of a program. If this is the case, simply hold down the CONTROL key and type the "1" key. The pause will continue until the CONTROL-1 key combination is repeated.

LIST can be used with optional parameters to display only a portion of the program. For example, LIST can be used with a single line number. LIST can also be used with a range of line numbers. The command LIST 10,30 would list all line numbers within the range 10 to 30, inclusive.

```
10 PRINT "Pat Kling"
20 PRINT "Rich Rheinhart"
30 PRINT "Karen Dorsey"
40 PRINT "Grady Dorsey"
LIST

10 PRINT "Pat Kling"
20 PRINT "Rich Rheinhart"
30 PRINT "Karen Dorsey"
40 PRINT "Grady Dorsey"

READY
LIST 10

10 PRINT "Pat Kling"

READY
LIST 10,30

10 PRINT "Pat Kling"
20 PRINT "Rich Rheinhart"
30 PRINT "Karen Dorsey"

READY
LIST 25,50

30 PRINT "Karen Dorsey"
40 PRINT "Grady Dorsey"

READY
```

Figure 3.4. Listing a program

EDITING A PROGRAM

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

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

The cursor can be moved by 4 of the keys on the right-hand side of the keyboard. These keys are labeled ↑, ↓, ←, and →. The "arrow" keys move the **cursor** in the direction of the arrow. The cursor is an inverse video rectangle that indicates the position where the next character entered via the keyboard will appear. Incidentally, the CONTROL key must be held down while the arrow keys are used.

Program lines can be manipulated by the INSERT, DELETE, and Back Space (Bk Sp) keys. The CONTROL key must be held down while using either INSERT or DELETE. The CONTROL-DELETE key combination deletes the character at the current cursor position. The CONTROL-INSERT key combination inserts a blank space into the program line. Pressing the Bk Sp key moves the cursor one position to the left and deletes the character in that position.

The use of the full screen editor is best explained using a simple example. Begin by entering the following program:

```
10 FOR S = 100 TO 200
20 PRINT "ANSWER IS   ";S
30 NEXT R
```

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

```
LIST

10 FOR S = 100 TO 200
20 PRINT "ANSWER IS   ";S
30 NEXT R

READY
```

Suppose that line number 30 was incorrect and was intended to appear as follows:

```
30 NEXT S
```

The correction can be made by using the ↑ key to move the cursor up to line 30. Proceed by pressing the → key until the cursor is on the "R". Correct the error by typing the correct letter, "S", then press RETURN.

Suppose that line number 10 was intended to read as follows:

```
10 FOR S = 1 TO 200
```

Use the arrow keys to place the cursor on the first offending "0". Pressing CONTROL-DELETE will remove the first "0" from the line. The cursor should now be positioned upon the other "0". Pressing CONTROL-DELETE a second time will delete the second "0". The cursor may also be placed on the zero on the end, and the Bk Sp key pressed to delete the unwanted zeros. Now, press RETURN. The Atari does not record any corrections until the RETURN key has been pressed. Therefore, once a line has been edited, always press RETURN to register the changes.

Finally, suppose line number 20 was also incorrect, and was intended to appear as follows:

```
20 PRINT "THE ANSWER IS  ";S
```

Use the arrow keys to place the cursor on the "A" in line 20. This is the position where additional characters are to be inserted. To insert four spaces into the line, press the INSERT key four times, while holding down the CONTROL key. Notice that while inserting, any characters to the right of the cursor will be moved over to make room for the additional spaces. Now, type the text to be inserted, THE. Remember to press RETURN after editing line 20, so that the changes will be stored.

RUNNING A PROGRAM

Once a program is present in memory, the operator can execute it. As mentioned previously, a program can be entered into memory via the keyboard or loaded into memory from a storage device -- cassette or disk. The procedure for loading a program will be discussed later in the chapter.

The RUN command is used to begin program execution. RUN can be used with or without an optional file specification as its parameter. Because RUN is generally executed without an optional parameter, the discussion of RUN in this section will be limited to its execution without a file specification. the usage of RUN with this parameter will be discussed in chapter 5.

When the RUN command has been entered and the RETURN key pressed, program statements entered in the indirect mode (with line numbers) will be executed in order, beginning with the lowest line. An example of the usage of RUN is shown in figure 3.5. The execution of a program can be stopped at any time by pressing the BREAK key, and resumed with the CONT command.

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

READY
```

Figure 3.5. RUN command

SAVING A PROGRAM

As you may recall from our discussion of program entry, only one BASIC program may be stored in memory at any one time. When the Atari's power is turned off, the contents of memory will be erased and any program stored there will be lost unless it is first stored on a permanent medium such as a diskette or a cassette tape.

Before a program can be saved on diskette, it must first be assigned a name from one to eight characters in length. This is known as a **filename.** Once a filename has been selected for a program, it can be stored using the SAVE command. The syntax of the SAVE command requires that the characters, D:, prefix the filename to indicate the disk drive. The filename and its prefix, together, are known as the **file specification.**

For example, if a program was presently residing in memory, it could be saved on a diskette with the following command:

```
SAVE "D:VAPNIK"
```

Notice that quotation marks are required around the file specification, D:VAPNIK.

When storing a program on cassette tape, no filename is required; however, a file specification is needed. The file specification for the cassette unit is C:.

```
SAVE "C:"
```

When the previous command is executed, a tone will sound twice as a signal to position the tape. At the tone, press the cassette unit's PLAY and RECORD keys. Finally, RETURN should be pressed on the Atari keyboard. Before pressing RETURN, the value of the tape counter should be written down so that the program may be easily found later. Incidentally, the CSAVE command may be used in place of the SAVE "C:" command with identical results.

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

Both cassettes and disks can be an effective means of retaining programs and data when the computer is turned off. The details of the procedures used to save programs and data will be presented in chapter 5.

LOADING A PROGRAM

Once a program has been saved on cassette tape or floppy disk, it can be loaded back into memory using the LOAD command. An example of a LOAD command is given below:

```
LOAD "D:VAPNIK"
```

Again, quotation marks are required around the file specification, D:VAPNIK.

If a file with the indicated filename cannot be found on the disk, an ERROR-170 (File not found) will be generated. Only files created with the SAVE command may be retrieved using LOAD. If an attempt is made to load a file not created with SAVE, an ERROR-21 (Bad load file) will be generated.

When retrieving a program from cassette, the file specification must be C:.

```
LOAD "C:"
(or)
CLOAD
```

When either of the previous commands is executed, a single tone will sound as a signal to position the tape (using the counter), and then press the cassette unit's PLAY key. Finally, RETURN should be pressed on the Atari keyboard.

LOAD automatically clears the Atari's memory before loading the designated program. Effectively, a NEW command is implied within the LOAD command.

MULTIPLE STATMENTS

In our examples thus far, only one BASIC statement has been included in each program line. In Atari BASIC, multiple statements may be included in a single program line as long as each statement is separated with a colon. Remember, however that the computer ignores any characters after a REM statement.

Thus:

```
10 PRINT "William":REM Bill:PRINT "NELSON"
20 END
```

will result in the output of;

```
William
```

Figure 3.6 uses valid multiple statements in line 10.

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

READY
```

Figure 3.6. Multiple statement lines

Data Types

The data processed in Atari BASIC can be classified under two special headings: string and numeric. String and numeric data are stored differently in memory by the Atari. Also, the various **operators** in BASIC affect string and numeric data in different manners. The two types of data will be described in the following sections.

STRINGS

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

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

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

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

"SEAN GRADY"
"12197"
"E97432"
"BOSTON, MA 01270"
"213-729-4234"

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

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

"Elaine said, "Goodbye," as she walked away."

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

NUMERIC DATA

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

Numeric constants consist of positive and negative numbers. Numeric constants cannot include commas. For example, 10900 would be a valid number in Atari BASIC, while 10,900 would be invalid.

Atari BASIC stores all numbers in memory using a **floating decimal point** form. Although all numbers are stored in the same form, they may be entered and displayed in one of two formats: **fixed point** or **floating point.**

Fixed point numbers can be defined as the set of positive and negative real numbers. Fixed point numbers include integers as well as numbers that contain a decimal portion. The following examples are numbers represented in fixed point notation:

+12383
-.007
36.2436
0
-14

Floating point numbers are represented in scientific notation. A number in scientific notation takes the following format:

$$\pm x \text{ E} \pm yy$$

$\pm$ is an optional plus or minus sign.

x is a fixed point number. This position of the number is known as the coefficient or mantissa.

E stands for exponent.

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

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

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

If a number can be expressed in scientific notation with an exponent less than -2 or greater than 9, scientific notation will be used to display the value. Otherwise, the fixed point notation will be used.

```
PRINT 1E8 <------ not greater
100000000          than 9

READY
PRINT .001
1.0E-3 <-------- less than -2

READY
```

Atari BASIC can only handle floating point numbers in the range between -9.99999999E+97 and +9.99999999E+97. Any decimal numbers in the range between -1E-98 and 1E-98 will be converted to zero.

Floating point numbers can have at most 9 significant digits. Any digits beyond 9 will be truncated.

```
PRINT 1E-99
0
READY
PRINT 1.234512345
1.23451234
READY
```

Variables -- An Overview

In the preceding section, we discussed BASIC's different types of data -- string and numeric. So far, data has only been represented as a constant. The value of a string or numeric constant such as "MICHELLE" or 382.436 always remains the same.

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

VARIABLE NAMES

BASIC allows variable names of any length. A variable name must begin with a letter of the alphabet followed by additional alphanumeric characters. Blank spaces are not allowed within a variable name. Only uppercase letters may be used in variable names; lowercase will not be accepted by the interpreter. The following are examples of valid BASIC variable names:

BENJI	X9
STASH	PHONE23

A variable name may duplicate a BASIC reserved word (see appendix C). However, the BASIC interpreter may be confused if a reserved word is used as a variable name. As a result, it is recommended that reserved words not be used as variables.

Variables, like constants, can either be string or numeric. Although BASIC automatically alots memory for numeric variables, the programmer must markedly reserve space for string variables. Memory is reserved using the DIM statement. The following example command would reserve room for up to 200 characters in the string variable, REBEL$. Incidentally, all string variable names must end in a dollar sign ($).

```
DIM REBEL$(200)
```

The following variable names would be declared as string and numeric, respectively. If a dollar sign is not included in the variable name, the variable is assumed to be numeric.

LOUISE$	BRIAN

ASSIGNMENT STATEMENTS

Numeric variables are initially assumed to have a value of zero. String variables are initially assumed to be null. Values may be assigned to a variable as the result of a calculation or as the result of an **assignment** statement. The reserved word, LET, is used to assign a value to a variable.

LET *variable = expression**

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

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

```
LET A = 5
    A = 5
```

LET is not useless, however. In cases where a reserved word is used as a variable name, LET serves to clarify the meaning of the program line.

```
COLOR = 3
ERROR -     COLOR■3

LET COLOR = 3
```

---

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

In the former of the preceding examples, the reserved word, COLOR, confused the interpreter causing an error. The LET in the latter example deciphered the meaning of the statement. Therefore, no error occured.

The value assigned to a variable can either be a constant, a variable, or the result of an operation. In the following example, A$ is assigned the string constant "JOHN". B is assigned the numeric constant 27.9. C is assigned the value of B, and D is assigned the numeric value of B multiplied by 2. Notice that the DIM statement in line 10 is required. This statement alots up to 20 characters for the string variable, A$.

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

READY
```

Variable types cannot be mixed. In other words, a numeric variable cannot be assigned a string value; nor can a string variable be assigned a numeric value.

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
14/7
3 * 1
A$ > B$
X AND Y
```

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

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

The operators in our previous examples were as follows:

```
+
/
*
>
AND
```

ARITHMETIC OPERATORS

Arithmetic operators are used to perform mathematical operations on numeric variables and constants. The various arithmetic operators are listed in table 3.1.

A number of these operations should already be familiar. The symbols + and - are used for addition and subtraction, respectively. The asterisk (*) is used to indicate multiplication, while the slash (/) is used to indicate division.

```
PRINT 5 + 3
8

READY
PRINT 24/8
3

READY
```

When the symbol "–" precedes a numeric constant or variable, it changes that value's sign. This usage is known as negation.

```
10 PRINT A = -5
20 PRINT A
30 PRINT -A
RUN
-5
5

READY
```

The second arithmetic operation specified in table 3.1 is **exponentiation.** Exponentiation (caret ^) is the process of raising a number to a specified power. For example, the following two expressions would evaluated identically as 125. The exponent, 3, indicates the number of times that the base, 5, is to be multiplied by itself.

```
5 ^ 3 = 125
5 * 5 * 5 = 125
```

Table 3.1. Arithmetic operations

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

ORDER OF EVALUATION (ARITHMETIC EXPRESSIONS)

The majority 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) + 3^2 * 2
A + B * A / (C + D)
27 + 47 / A ^B
```

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

In this section, the order of evaluation of compound arithmetic expressions will be discussed. Later in this chapter, the order of evaluation of relational and logical operators will be discussed. Also, the relative evaluation priorities of these three groups will be will be outlined.

In an expression with more than one arithmetic operator, the operators with higher priorities are evaluated first followed by those with lower priority. If two operators have the same priority, evaluation is performed from left to right in the expression. The operators in table 3.1 are listed in descending priority. For example, exponentiation is listed before multiplication, because exponentiation has a higher priority. Multiplication and division have the same priority. Also, addition and subtraction have the same priority. The following is an example of the evaluation of the arithmetic operators in an expression:

```
A = 37.1 + 12.9 * 2.1 - 7 + 4 ^2
  = 37.1 + 12.9 * 2.1 - 7 + 16
  = 37.1 + 27.09 - 7 + 16
  = 64.19 - 7 + 16
  = 57.19 + 16
  = 73.19
```

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

```
A = (37.1 + 12.9) * 2.1 - (7 + 4^2)
  = 50 * 2.1 - (7 + 16)
  = 50 * 2.1 - 23
  = 105 - 23
  = 82
```

RELATIONAL OPERATORS

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

<	less then
<=	less then 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 1.5 was compared to the constant 2.4 to see whether they were equal, the expression would evaluate to false. In 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). The values can be used as any other numeric expression would be used. The following relational expressions and their results demonstrate comparisons:

```
PRINT 5 > 7
0          <-------- false

READY
PRINT 5 > 3
1          <-------- true

READY
PRINT 7 = 7
1          <-------- true

READY
```

Relational operations are evaluated after the arithmetic operations. Relational operations are performed from left to right in an expression — each having the same priority.

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

For example, consider the two string values "BONNIE" and "BECK". In a relational expression, the initial characters of the strings will be compared first. Since both strings begin with "B", the comparison will continue with the second character. Since the ASCII code for "E" (69) is less than the ASCII code for "O" (79), "BECK" is considered less than "BONNIE".

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

Blank spaces are counted in string comparisons and have an ASCII value of 32. Lowercase letters have higher ASCII values than uppercase letters. Therefore, "Z" is less than "a". Appendix B lists the various string characters and their corresponding ASCII values.

The following examples demonstrate the use of relational operators with string values. All of the following expressions are true. Notice that all string constants must be enclosed in quotation marks.

```
"LORRIE" = "LORRIE"
"LORRIE" > "LAURIE"
"PAT" < "PATRICK"
"PAT RICK" < "PATRICK"
"elaine" > "MOST"
A$ > Z$ where A$ = "elaine" and Z$ = "MOST"
```

LOGICAL OPERATORS

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

The logical operators are NOT (logical complement), AND (conjunction), and OR (disjunction). The results of the logical operators are summarized in figure 3.7. These charts are known as truth tables.

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. A value of 1 corresponds to a true result, while a value of 0 is considered false. This value may be used as would any other numeric value.

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

```
PRINT 1 + 1 = 2 AND 1 + 1 = 3
0

READY
PRINT 1 + 1 = 2 AND NOT(1 + 1 = 3)
1

READY
PRINT -5 OR NOT 3
1

READY
```

In the first example, the result of the logical expression was false. Although 1 + 1 = 2 is true, 1 + 1 = 3 is not true (false). In the second example, 1 + 1 = 3 is again false; therefore, NOT (1 + 1 = 3) is true. Since both expressions of the logical operator are true, the entire expression is true. In the final example, NOT 3 is false. (3 is non-zero; therefore, it is true.) Likewise, -5 is true because it is non-zero. Since one of the arguments of the OR operator is true, the entire expression is true.

Both the relational and logical operators are generally used in the context of an IF...THEN statement. Here, program flow may be influenced depending on whether an expression evaluates to true or false.

```
IF X > 10 OR Y < 0 THEN 900
```

In the previous example, the result of the logical operation will be true if the variable X is greater than 10 or if the variable Y is 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.

NOT Operation

X	NOT X
true	false
false	true

OR Operation

X	Y	X OR Y
true	true	true
true	false	true
false	true	true
false	false	false

AND Operation

X	Y	X AND Y
true	true	true
true	false	false
false	true	false
false	false	false

Figure 3.7. Logical operators

OVERALL ORDER OF EVALUATION

In this chapter. the use of the arithmetic, relational, and logical operators have been outlined. Table 3.2 summarizes the evaluation order of these operators. The single exception to the rules given by the table occurs when the relational operators are used to compare strings. In this case, the relational operators are given the highest priority.

Incidentally, a unary operator can be defined as an operator having a single expression for its argument. NOT is the logical unary operator, while "-" is the arithmetic unary operator.

Table 3.2. Overall order of evaluation

		Symbol	Priority
Unary	Negation Logical Complement	— NOT	1
Arithmetic	Exponentiation	∧	2
	Multiplication Division	* /	3
	Addition Subtraction	+ -	4
Relational	Equality Inequality Less than Greater than Less than or equal to Greater than or equal to	= <> < > <= >=	5
Logical	Conjunction	AND	6
	Disjunction	OR	7

4

BASIC Programming Concepts

Introduction

In chapter 3, BASIC programming fundamentals were discussed. In this chapter, we will explain some additional fundamental programming concepts. These include:

- data input and output
- conditionals, branching and loops
- tables and arrays
- functions
- string handling
- program chaining

Inputting and Outputting Data

Thus far, we have briefly described the usage of the PRINT statement to output data. Now, we will discuss the usage of PRINT to format the outputted data. After we have discussed the methods used to output data, we will discuss the statements used to input data into variables. These include INPUT and GET.

PRINT

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

```
PRINT "PAT";"MIKE";"KEN";"JOHN"
PATMIKEKENJOHN
READY
PRINT "PAT","MIKE","KEN","JOHN"
PAT     MIKE     KEN     JOHN
READY
```

In the first example, the semicolon was used as the delimiter. The semicolon causes each string data item in the PRINT statement to be output immediately adjacent to the preceding item.

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

```
10 DIM A$(5)
20 DIM B$(5)
30 A$ = "ATARI"
40 B$ = "130XE"
50 PRINT A$;B$
60 PRINT A$;" ";B$
RUN
ATARI130XE
ATARI 130XE
```

In the second example on page 92, comma's were used to delimit the string constants. 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.

The number of spaces in each print zone can be changed by placing a new value into memory location 201. For example, the statement,

```
POKE 201,20
```

would cause each print zone to contain 20 spaces.

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

```
100 POKE 201,20
200 PRINT "Name","ID No."
300 PRINT "Diana Growski","0-4377"
400 PRINT "Tim Mirroli","F-0010"
500 PRINT "Mary Bungalow","B-8008"
600 POKE 201,10
700 END
RUN
Name                ID No.
Diana Growski       0-4377
Tim Mirroli         F-0010
Mary Bungalow       B-8008
READY
```

The POKE statement in line 100 causes each print zone to consist of 20 spaces. Lines 200 through 500 display data on the screen using commas as delimiters. Line 600 causes the print zones to consist of 10 spaces.

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

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

```
10 PRINT "DATA1";
20 PRINT "DATA2";
30 PRINT "DATA3";
40 END
RUN
DATA1DATA2DATA3
READY
```

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

```
10 PRINT "DATA1",
20 PRINT "DATA2",
30 PRINT "DATA3",
40 END
RUN
DATA1     DATA2     DATA3
READY
```

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"EE↑EE↑EE↑"
200 END
RUN
↑↑↑
```

In this example, $^{E}{}_{E}$ represents pressing the Escape key twice, and ↑ represents pressing Escape Ctrl--. The escape sequences are given below.

Table 4.1. Escape Sequences

Keyboard Entry	ASCII Code	Echoed Character	String Character
ESC/ESC	27	$^{E}{}_{E}$	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.

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			

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

```
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 figure 4.1 when it is run.

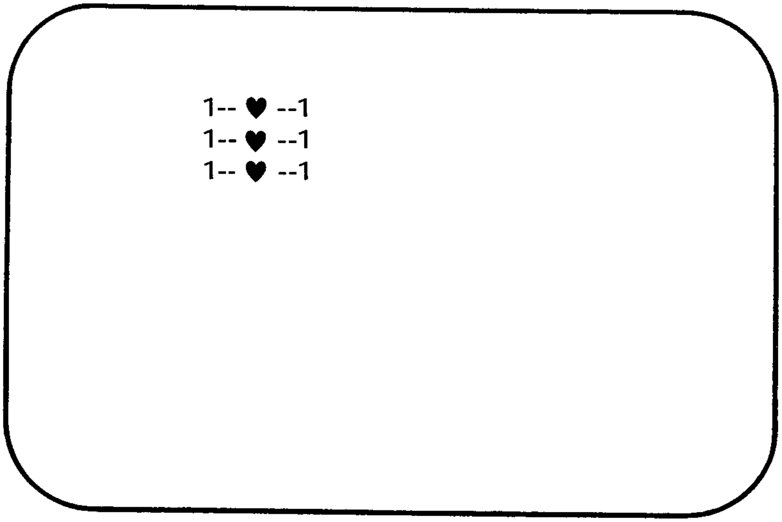


Figure 4.1. Graphics example program output

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.

---

* ♥ -- is generated by pressing Ctrl-.

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 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 a 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 fifth column.

A tab stop can be cleared in the immediate mode by moving the cursor to the position desired 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 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 sequences 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 right each time it is pressed.

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

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

the string variable A$ is set equal to the cursor down character set. 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 this 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 Statements

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.

Remember, rows are numbers 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 left or right margins. The following statments 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.

INPUT

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

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

```
100 DIM A$ (255),B$ (255)
200 INPUT A$, B$,C,D
```

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

```
JOHN    *
SMITH   *
281,347  *
```

These entries will be assigned to the variables as follows:

```
A$ = "JOHN"
B$ = "SMITH"
C = 281
D = 347
```

A potential problem arises when using numeric variables within an INPUT statement. If a string constant is input for a numeric variable, the following error would be displayed:

```
ERROR — 8 AT LINE 200
```

---

* denotes pressing the RETURN key.

and the computer will cease execution of the program.

In many cases, it is a good idea to use only string variables in an INPUT statement. Once a string has been entered through the INPUT statement, it can be converted to its numeric equivalent by using the VAL function. The VAL function will be explained in detail later in this chapter.

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 program lines:

```
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 ranges, string entries that are longer than allowed for by the INPUT statement's variables, and an input response other than that prompted for.

The very nature of the INPUT statement prevents certain errors from occuring as these are detected by the BASIC interpreter. For example, 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. All of these statements will be covered later in this chapter.

For example, in the following program, the operator's input is checked with two IF-THEN statements. If the response is not one 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
```

GET

The GET statement, like the INPUT statement, is used to enter data. The difference between GET and INPUT is that GET will accept only one character per entry. This is convenient when a single character response is needed in a program.

The GET statement must be used in conjunction with the OPEN statement. The OPEN statement must open a channel from the keyboard. The following configuration is used to open a channel from the keyboard:

OPEN #*filenumber*,4,0,"K:"

The *filenumber* may be any integer from 0 to 7.

Once the channel from the keyboard is opened, the GET statement may be used. The GET statement uses the following configuration:

GET #*filenumber, numeric variable*

The *filenumber* must be the same as that specified in the OPEN statement used to open the keyboard. The ASCII code of the character that is entered will be assigned to the numeric variable.

When a GET statement is encountered, the computer will wait for one key to be pressed. When a key is pressed, the ASCII code of that character will be assigned to the numeric variable and program execution will continue. The following example shows the use of a GET statement.

```
100 OPEN #1,4,0,"K:"
200 PRINT "DO YOU WISH TO CONTINUE(Y/N)?"
300 GET #1,A
400 IF A = 89 OR A = 121 THEN 700
500 IF A = 78 OR A = 100 THEN 900
600 GOTO 300
700 PRINT "YOU PRESSED Y FOR YES"
800 END
900 PRINT "YOU PRESSED N FOR NO"
1000 END
```

Conditionals, Branching and Looping

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

IF-THEN	ON-GOTO
GOTO	ON-GOSUB
GOSUB	TRAP
FOR-NEXT	

These statements will be discussed in the following sections.

CONDITIONALS

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

IF *expression* THEN *statement*

The IF statement sets up a decision. If *expresssion* evaluates to true, then *statement* will be executed. If *expression* evaluates to false, the subsequent program statement will be executed. In the following example, if AGE is greater than or equal to 21. "LEGAL" will be printed.

```
IF AGE > = 21 THEN PRINT "LEGAL"
```

BRANCHING STATEMENTS

Branching statements change the execution pattern of programs from their usual line by line execution. 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*

The following program shows the effect of the GOTO statement.

```
10 PRINT "THIS IS LINE 10"
20 PRINT "THIS IS LINE 20"
30 GOTO 50
```

program continued on next page

```
40 PRINT "THIS IS LINE 40"
50 PRINT "THIS IS THE END"
60 END
RUN
THIS IS LINE 10
THIS IS LINE 20
THIS IS THE END
READY
```

In the preceding program, the GOTO statement in line 30 transferred program control to line 50. The GOTO statement can be used to transfer program control to any line within a program.

SUBROUTINES AND GOSUB

Many times you will find that the same set of program intructions 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 *linenumber*

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

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

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

CONDITIONAL STATEMENTS WITH BRANCHING

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

```
100 DIM A$(10)
200 PRINT "ENTER THE AMOUNT";
300 INPUT A
400 IF A = 0 THEN 700
500 PRINT A
600 GOTO 200
700 PRINT "ARE YOU FINISHED";
800 INPUT A$
900 IF A$ <>"Y" THEN 200
1000 END
```

In our preceding example, if the value input for A has a zero value, then the program will branch to line 700 where the operator will be asked whether he has finished entering data. In line 900, the program will set up a condition where if the input was anything other than the letter "Y", the program will branch to line 200. If the entry was equal to Y, the program will end at line 1000.

Note in line 900 that a GOTO statement is not used to precede the line number being branched to. When a line number is indicated following a THEN statement, the computer assumes the presence of GOTO.

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

```
100 INPUT A
200 ON A GOTO 400,600
300 GOTO 999
400 PRINT "A = 1"
500 GOTO 999
600 PRINT "A = 2"
999 END
```

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

If the variable or expression evaluates to a number greater than the number of line numbers following GOTO program control will branch to the statement immediately following the ON-GOTO statement. This is also the case if the variable or expression following ON evaluates to zero. Negative values and values greater than 255 are not allowed for the control expression.

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

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

If the value of X is 1, the subroutine at line 1000 will be executed. If X is 2, the subroutine at line 2000 will be executed. If X is 3, the subroutine at line 3000 will be executed. If X evaluates to 0 or to a number between 3 and 255, the statement immediately following the ON-GOSUB will be executed. If X evaluates to a negative value or a value greater than 255, an error will occur.

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

LOOPING STATEMENTS

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

```
100 A = 1^2
200 PRINT A
300 B = 2^2
400 PRINT B
500 C = 3^2
600 PRINT C
```

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

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

The sequence of statements from line 100 to 400 is known as a **loop**. When the computer encounters the FOR statement in line 100, the variable A will be set to 1. X will then be 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 will be incremented (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 increased 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 increased. The integer appearing after STEP is the new increment. For example, if our preceding example were changed as follows:

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

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

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

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

Our preceding example is used to read data into the numeric array R. Arrays, as well as the READ and DATA statements, will be discussed in detail later in this chapter.

Be certain that any inner loop is ended prior to ending its outer loop. Also, be certain that every NEXT statement has a matching FOR statement. If the BASIC interpreter cannot match every NEXT statement with a preceding FOR statement, an error will result.

ERROR HANDLING

In some situations, it is easier to correct problems as they occur in a program, rather than avoid them. This technique is called **error handling**. BASIC allows the use of a TRAP statement to specify a line number where a program should proceed if an error occurs. This feature allows a portion of the program to be set aside as an error handling routine. Error handling routines are commonly used to correct small problems that occur infrequently in a program.

The following program demonstrates the technique used to branch a program in event of an error:

```
10  TRAP 100
20  PRINT "INPUT X";
30  INPUT X
40  Y = X^0.5
50  PRINT "The square root of",X,"is";Y
60  END
100  Y = (-X)^.5
110  PRINT "The square root of";X";is";Y;"i"
120  END
```

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

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

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

It is not necessary to understand the use of "complex" numbers to comprehend the example. The main concept of the program is:

> The statement at line 40 would normally have caused an error if a negative value had been input for the variable X. However, in this case, the TRAP statement causes the program to branch to line 100 whenever an error occurs.

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

```
RUN
INPUT X:4 <------------------ user's response
The square root of 4 is 2
READY
RUN
INPUT X:-16 <---------------- user's response
The square root of -16 is 4i
READY
```

There are several memory locations that are used to store information regarding the error which has occured. Memory location 195 stores the error code of the previous error. Also, memory locations 186 and 187 can be used to determine the line number where the error occured. The following example shows how these memory locations can be used in a program.

```
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)
RUN
?JOHN <-------------- user's response
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 occured is displayed. The following expression:

```
256*PEEK(187)+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 a 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).

Appendix A contains the BASIC error messages along with their corresponding error numbers and descriptions of the errors.

Tables and Arrays

In chapter 3 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.

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

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 or disk, the variable name table is saved along with the program. If the program is later loaded back into memory with the LOAD or CLOAD statement, the variable name table will also be read into memory and will take the place of the existing variable name table.

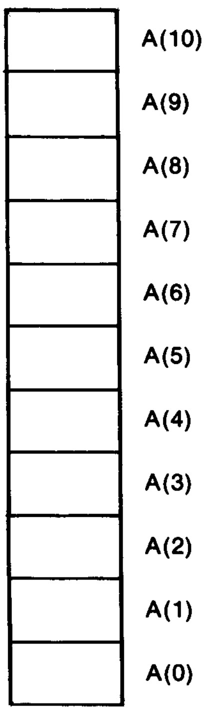
SUBSCRIPTED VARIABLES

Obviously, the use of thousands of individual 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(10)

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

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



Arrays can have up to two dimensions. Two-dimensional arrays are also known as tables. A table containing 6 rows and 8 columns is depicted below:

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

Array variables can be assigned values and used with most operators. However, array variables cannot be used in a READ, INPUT or GET statement. The following 2 examples illustrate the use of array variables.

Example 1

```
10 DIM A(5)
20 A(0) = 5
30 A(1) = 6
40 A(2) = 7
50 A(3) = 8
60 A(4) = 9
70 PRINT A(0)*A(1)
80 A(5) = A(2) + A(4)
90 PRINT A(5)
100 END
RUN
 30
 16
READY
```

Example 2

```
10 DIM A(7)
20 FOR J = 0 TO 6
30 READ D
40 A(J) = D
50 NEXT J
60 FOR J = 0 TO 6
70 PRINT A(J)
80 NEXT J
90 DATA 10,15,8,14,14,9,14
```

DIMENSIONING AN ARRAY

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

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

```
DIM A(20)
```

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

Two dimensional arrays are dimensioned as follows:

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

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

Generally, it is good programming practice to group all DIM statements at the beginning of the program. This prevents an array variable from inadvertently being referenced before it has been dimensioned. Referencing an array variable before it has been dimensioned will result in an error.

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

```
10 PRINT FRE(0)
20 DIM A(25,25)
30 PRINT FRE(0)
40 CLR
50 PRINT FRE(0)
60 END
RUN
13242
9186
13242
READY
```

In line 10, the number of available bytes in memory is displayed. FRE is a function which displays the available free bytes in memory. FRE is explained in more detail, later in this chapter.

In line 20, the DIM statement reserves an area in memory for a table consisting of 676 elements. From line 30, it is evident that the number of free bytes has decreased substantially. This is due to the fact that an area of memory has been reserved for the elements in table A.

In line 40, the CLR statement reverses the DIM statement and frees the memory previously required for the elements in table A. The CLR statement is also a memory management command.

DATA & READ STATEMENTS

Earlier, we discussed how data could be assigned to a variable with a LET statement as well as how data could be input directly from the keyboard and assigned to a variable with an INPUT statement. However, none of these statements are practical for assigning data values to the individual variables in a large array or table. DATA and READ statements are much more practical for assigning values to variables in an array.

A typical DATA statement is shown below:

```
100 DATA WILLIAMS,CLEVELAND,OHIO,44109
```

Notice that this DATA statement contains four data items, three of which are string, and one of which is numeric. Notice also that the string data items need not be enclosed in quotation marks. A data item is determined as string or numeric depending on the variable type in the

READ statement.

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

```
200 READ NAME$,CITY$,STATE$,ZIP
```

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

If the first DATA statement encountered does not have enough data items to be assigned to all the variables in the READ statement, the next DATA statement will continue to be assigned to the variables in the READ statement until all of the variables in the READ statement have been assigned a value.

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

BASIC includes a statement known as RESTORE, which when executed, sets the DATA item pointer back to the beginning of the DATA statement list. The use of the DATA item pointer and the effect of RESTORE on it is depicted in figure 4.2.

The RESTORE statement may be used with a line number following the reserved word. When a RESTORE is used in this manner, the DATA item pointer is set to the first item of the DATA statement in that line. For example, if line 400 in our example had the following.

```
400 RESTORE 110
```

the READ statement in line 500 would have assigned the value 27 to the variable X.

When not properly used, DATA and READ statements can be the source of program errors. One potential error source occurs when the

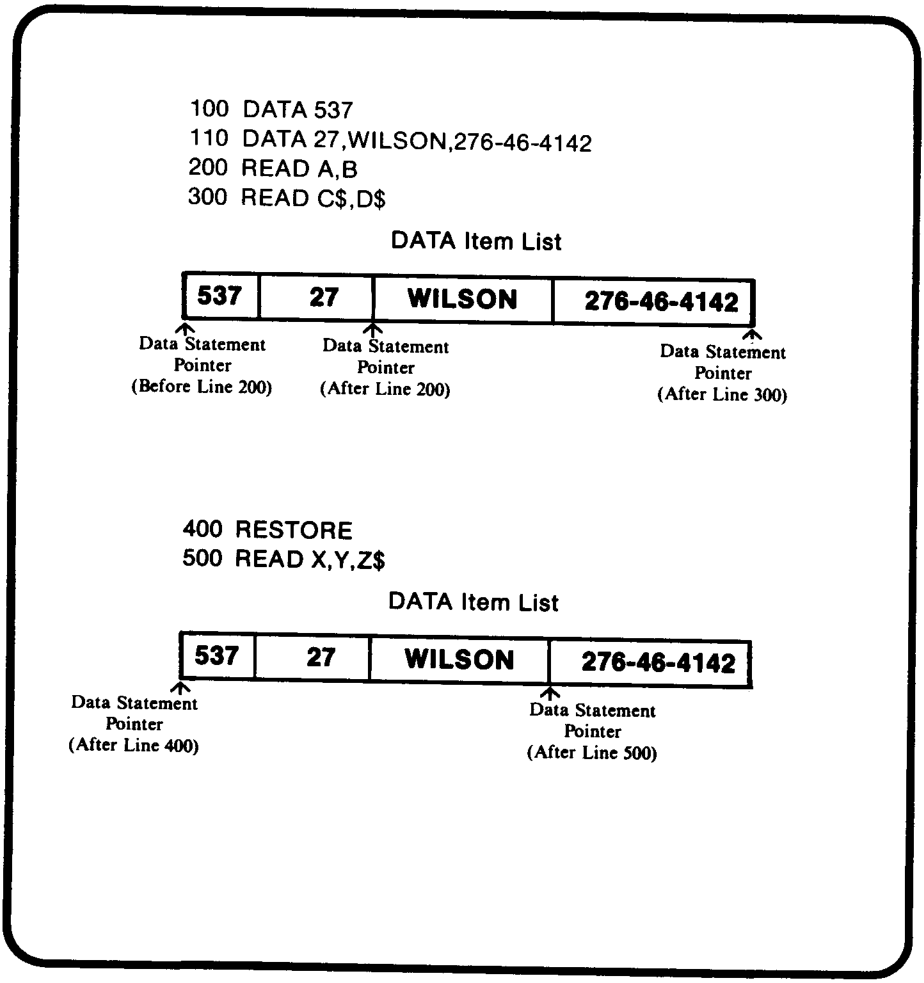


Figure 4.2. DATA item pointer

program attempts to READ more data items than were given in the DATA statements. Such an error would occur in the following program:

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

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

```
ERROR—    6 AT LINE 300
```

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

```
ERROR—    8 AT LINE 300
```

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

```
10 DIM YEAR(7),INCOME(7)
20 PRINT "YEAR",,"INCOME"
30 FOR K = 0 TO 6
40 READ Y,I
50 YEAR(K) = Y
60 INCOME(K) = I
70 NEXT K
80 FOR K = 0 TO 6
90 PRINT YEAR(K),,INCOME(K)
100 NEXT K
110 DATA 1978,20876
120 DATA 1979,21456
130 DATA 1980,21987
140 DATA 1981,22396
150 DATA 1982,22987
160 DATA 1983,24098
170 DATA 1984,25234
RUN
YEAR          INCOME
1978          20876
1979          21456
1980          21987
1981          22396
1982          22987
1983          24098
1984          25234
```

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

```
10 DIM A(3,4)
20 DATA 10,20,30,40
30 DATA 50,60,70,80
40 DATA 90,10,20,30
50 FOR J = 0 TO 2
60 FOR K = 0 TO 3
70 READ D
80 A(J,K) = D
90 NEXT K
100 NEXT J
110 FOR J = 0 TO 2
120 FOR K = 0 TO 3
130 PRINT A(J,K)
140 NEXT K
150 PRINT
160 NEXT J
170 END
RUN
10     20     30     40
50     60     70     80
90     10     20     30
READY
```

The preceding program would read data items into table A( ) as shown in the following illustration.

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

Functions and String Handling

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

In BASIC, a number of functions are already defined by reserved words and are a part of the BASIC interpreter. These are known as **built-in** functions (see table 4.3). Built-in functions cover a wide range of standard math operations such as absolute value, square root, logarithms, etc. Built-in functions are also available for working with strings, as well as a variety of other operations. Both numeric and string functions will be discussed in this section.

BUILT-IN MATHEMATICAL FUNCTIONS

The majority of BASIC functions are used in mathematical applications. We provide an overview of BASIC's math functions in this section. Each individual function will be described at the end of the chapter.

Table 4.3. BASIC built-in functions

ABS	DEG	PEEK	STICK
ASC	EXP	PTRIG	STRIG
ADR	FRE	RAD	STR$
ATN	INT	RND	USR
CHR$	LEN	SGN	VAL
CLOG	LOG	SIN	
COS	PADDLE	SQR	

All of the BASIC mathematical functions operate in much the same manner. Each function is defined by a reserved word (ex. SIN for sine, COS for cosine, LOG for logarithm etc.).

A numeric constant, variable, or expression may appear in parentheses following the reserved word which identifies the function. The function for that numeric value will then be calculated by the computer. The use of several mathematical functions is shown in figure 4.3.

BASIC includes the following three trigonometric functions:

SIN(N) = sine of the angle N.
COS(N) = cosine of the angle N.
ATN(N) = arctangent of the angle N.

The angle N may be given in either radians or degrees. The two commands, DEG and RAD, are used to specify whether the angle is going to be in degrees or radians.

Executing the DEG command will cause any subsequent trigonometric functions to treat their arguments as degrees. Executing the RAD command causes any subsequent trigonometric functions to treat their arguments as radians.

When the system is powered up, or when a NEW or RUN command is issued, the computer defaults to radians.

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

A = EXP(B)

The preceding EXP function is calculated by computing the value of **e** raised to the B power, **e** is known as the base of natural logarithms. The value **e** in BASIC is 2.7182179.

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

LOG(X) = natural logarithm of X

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

$\mathrm{LOB}_b(X) = \mathrm{LOG}(X)/\mathrm{LOG}(b)$

where b is the base of the logarithm.

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

```
100 PRINT SIN(0.47)
200 PRINT COS(0.98)
300 PRINT ATN (0.38)
400 PRINT SQR(49)
500 PRINT INT(5.79)
600 PRINT INT(-5.79)
```

program continued on next page

```
700 PRINT ABS(-4.7)
800 PRINT SGN(2.7)
900 PRINT SGN(-2.7)
1000 END
RUN
0.452886286
0.557022556
0.363147009
7
5
-6
4.7
1
-1
READY
```

Figure 4.3. Mathematical functions

SQR(X) = positive square root of X

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

X ^ (1/2)

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

X ^ (1/3)

BASIC also includes several functions that can be used in working with numeric values. These include INT, ABS, and SGN. The INT function returns the integer with the greatest value which is less than or equal to its argument. INT takes the following form:

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

Figure 4.3 contains examples of the usage of the INT function.

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

ABS(X) = |X|

An example of the use of ABS appears in figure 4.3.

The SGN function returns the sign of its argument. An example of the use of SGN appears in figure 4.3.

STRINGS & STRING HANDLING

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

JOSEPH IZAK

the substring defined by X$(1,6) would consist of "JOSEPH", and X$(8,11) would consist of "IZAK". 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. It if 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 "TIM MIRROLI" to "DON MIRROLI".

```
100 DIM X$(15)
200 X$ = "TIM MIRROLI"
300 PRINT X$
400 X$(1,3) = "DON"
500 PRINT X$
600 END
```

Line 100 in the previous program dimensions X$ to 15. Line 200 assigns the string "TIM MIRROLI" to X$. Line 300 prints X$. Line 400 replaces the characters 1 through 3 in X$ with "DON". Line 500 prints the new X$.

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*)

Atari BASIC uses the following configuration for string concatenation.

Variable1$*(LEN(*variable1$)+1)=*variable2$*

Variable1$* and *variable2$ are both string variables. The contents of *variable2$* will be joined to the contents of *variable1$*. The entire string will be assigned to *variable1$*.

The following program will illustrate string concatentation in Atari BASIC.

```
100 DIM X$(10)
200 DIM Y$(10)
300 X$ = "JOHN"
400 Y$ = "NIN"
500 X$(LEN(X$)+1)=Y$
600 PRINT X$
700 END
RUN
JOHNNIN
```

The actual concatenation takes place in line 500. 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 following the end of the original X$.

If line 300 was revised as follows,

```
300 X$ = "JOHN "
```

the following could be output:

```
JOHN NIN
```

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

STRING/NUMERIC DATA CONVERSION

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

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

Examples of the use of STR$, VAL, CHR$, and ASC are given in figure 4.4 and figure 4.5.

```
100 DIM W$(15),X$(15)
200 W = 12345
300 W$ = STR$(W);REM W$ = "12345"
400 X = 6789
500 X$ = STR$(X);REM X$ = "6789"
600 W$ = (LEN(W$)+1) = X$;REM W$ = "123456789"
700 W = VAL(WS);REM W = 123456789
800 Z = W/1000
900 PRINT Z
1000 END
123456.789

READY
```

Figure 4.4. Use of STR$ and VAL

```
100 DIM A$(15),X$(15)
200 A$ = "GEORGE"
300 A = ASC(A$)
400 PRINT A
500 X = 90
600 X$  CHR$(90)
700 PRINT X$
800 END
RUN
71
Z
```

Figure 4.5. Use of ASC and CHR$

Program Chaining

The final topic covered in this chapter is program chaining. A long program may overrun the memory of the Atari. When this is the case, the program can be separated into two or more self-sufficient parts. If a portion of the program is needed that is not currently in memory, it can be loaded and executed by the RUN command.

The RUN statement can be included as a program line in one program in order to load and execute another program. 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.

5

File Handling

Introduction

In the preceding chapters, we did not discuss the concepts and programming techniques related to storage of data on cassette tape or diskette. In this chapter, these concepts will be discussed. The writing of programs that make use of these devices will also be discussed.

Files, Records and Fields

Before learning specific concepts which relate to the cassette tape unit and diskette drives, it is essential that the user understand the concepts of **files, records,** and **fields**.

A file can be defined as a collection of related data. Files can be distinguished as being either **program files** or **data files**. A program file consists of a program which has been saved on diskette or cassette tape.

A data file consists of a collection of related information which has been saved on a diskette or cassette tape. Generally, a data file is read from storage by a program or written to storage by a program. Data files are divided into smaller segments known as records and fields. A field is a single piece of data. Fields are grouped together as a record. These records, in turn, make up the file

A simple illustration may help you understand the concepts of a data file, record, and field. Take an address book as an example of a data file. This file would contain name, address, and telephone number data for the individuals appearing in the address book. Each individual's name, address, and phone number would represent one record. For example, the following data would make up one record:

Jay Gatsby
1 Shore Lane
West Egg, NY 10565
516-787-2122

Each individual data item within the record (i.e. name, street address, city, state, zip code, telephone number) could be thought of as a field.

A data file is written or read as a series of constants. For example, our address book example might be read as follows:

```
"Jay Gatsby","1 Shore Lane","West Egg","NY",10565,"516-787-2122"
"Nick Carraway","7 Shore Lane","West Egg","NY",10565,"516-787-2736"
```

When these data items are read or written, the first field will have been defined as the name, the second as the street address, the third as a city, the fourth as the state, the fifth as the zip code, and the sixth as the telephone number.

Note that the fifth field is numeric, while the others contain string data. Notice that the string data is enclosed in quotation marks. Finally, note that each data item is separated by a comma. For the computer to be able to distinguish where one data item ends and another begins, these items must be separated with a character known as a **delimiter**. A delimiter might be a comma (as in our example), a blank space, a line return character, or a form feed character.

The advantages of using data files with programs is obvious. Data files allow the user to save, alter, and redisplay data as is necessary. For example, using our address book as an example, programs could be written to do the following.

1. Enter changes in an individual's record by reading the file from storage until the desired record is found, inputting the required changes, and rewriting the file back into storage.
2. Displaying an individual's name, address, and telephone information by reading the file from storage until the desired record is found, outputting the field data to the screen and rewriting the file back into storage.

The use of a data file with a mass storage device is analogous to the use of a file cabinet for storing information in an office.

FILE SPECIFICATIONS

Every file is identified with a **file specification** that consists of a **filename** and a **device name**. The filename identifies which file to search for, while the device name identifies where the file can be located.

```
C: ACCOUNT
D1: GAMES
D4: BOWLING.SCR
```

Because only the disk drive device can access more than one file at any one time, a filename is only required when using the disk drive. A filename is optional when using any other device, and, in this case, serves only as a memory aid to the programmer.

A filename can include up to eight alphanumeric characters. In other words, the only characters that can be used in a filename are the letters **A** through **Z**, and the numbers **0** through **9**. A filename may also include an optional three character extender. A **filename extension** consists of a period and three alphanumeric characters which appear immediately after the primary filename.

A filename can be entered with uppercase or lowercase letters. The computer will interpret all lowercase entries as capitals. The following filenames all refer to the same file:

Holyname.HS
HOLYNAME.hs
HOLYNAME.HS
holyname.hs

The file specification prefixes the filename with a device name. The device name designates the storage device that is to hold the file. A device name can consist of one or two characters followed by a colon. Table 5.1 lists the device names that are recognized by the Atari. Of course, any required hardware must be included in the system for a device to operate correctly.

Each of the devices in the table can accept files.Although the disk drive and cassette unit will be used extensively in this chapter as both input and output devices, any of the devices listed in Table 5.1 could easily be substituted. A few of the devices can only be used for input or output. For example, P: can only be used as an output file, while K: can only be used as an input file.

Table 5.1. Input/output devices

Device Name	Reference	Input	Output
C:	Cassette	x	x
D1: or D:	First Disk Drive	x	x
D2:	Second Disk Drive	x	x
D3:	Third Disk Drive	x	x
D4:	Fourth Disk Drive	x	x
E:	Screen Editor	x	x
K:	Keyboard	x	
P:	Printer		x
R1: or R:	RS232 Port #1	x	x
R2:	RS232 Port #2	x	x
R3:	RS232 Port #3	x	x
R4:	RS232 Port #4	x	x
S:	Display	x	x

File Access

File access refers to the process of reading data from a file or writing data to it. In BASIC, data is organized in a file in either a sequential or a random manner. The mode in which a file's data is organized determines how that data will be accessed. Random access does not mean that the file is stored in a haphazard manner. Random access denotes that any part of the file can be accessed directly. Sequential access denotes that the file's data must be read or written in a specific order.

SEQUENTIAL AND RANDOM FILES

Two types of data files are used in Atari BASIC, sequential data files and random access data files. All of the aforementioned devices support sequential files, while the disk drive also allows random files.

Each record of a sequential disk file is assigned exactly as much storage space as it requires. There are no blank spaces between records in a sequential file. In random data files, a constant space is assigned to every record in the file. If the record does not occupy the entire space assigned to it, the remaining space is left blank.

The concepts of sequential and random files are pictured in figure 5.1. Notice that the length of each record in the random file is constant at 100 bytes. The record length of a sequential file is variable.

Random File

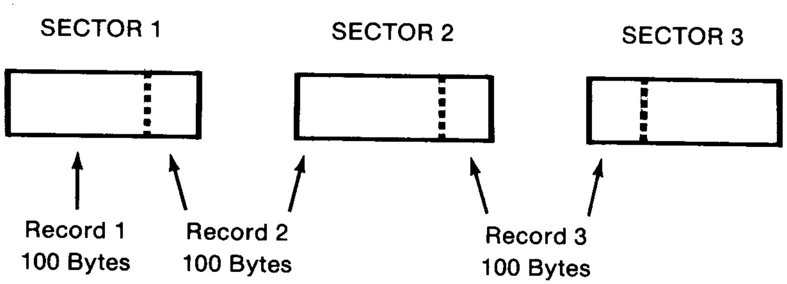


Figure 5.1 continued on next page

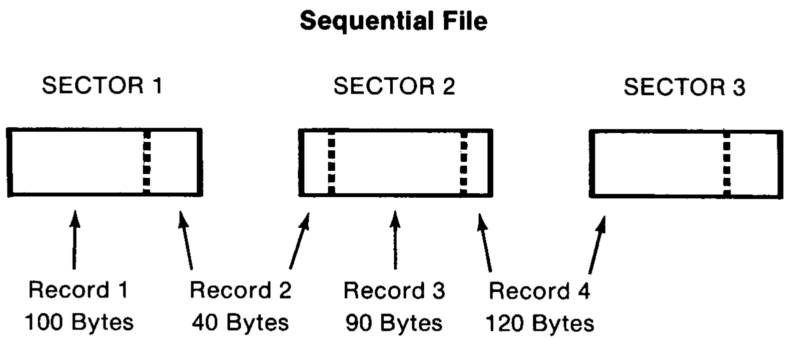


Figure 5.1. Random and sequential data files

The important difference between random and sequential files lies in how each file is accessed. **Direct access** of any record in a random file is possible regardless of where that record is located in the file. By direct access, we mean that any record in the file may be retrieved regardless of its position, without having to search through the entire file to find it. With random files, BASIC knows the length of each record and can easily calculate the location of any record in the file.

Records in a sequential file can only be retrieved by **sequential access**. In sequential access, the record search begins with the first record in the file and must continue until the desired record is found. In other words, to find record 17 in a sequential file, BASIC would have to read the preceding 16 records. BASIC has no way of determining the location of record 17, other than reading the first 16 records.

Both random and sequential files have advantages and disadvantages. Sequential files use less disk space than do random files. Each record in a sequential file is assigned only the disk space that it needs. Random files require every record to be the same length. Therefore, each record must be assigned the amount of disk space required by the longest record in the file. Generally, this results in wasted space.

Random files have an advantage over sequential files in that a record from a random file may be read into memory, changed, and then written back to the disk without affecting the rest of the file. Record editing, however, cannot be done in a sequential file because any change in a single record's length will adversely affect the entire file.

Opening a Sequential File

Before a file can be read from or written to, it must be opened. When a file is opened, first the operating system is called upon to obtain information from the disk regarding the file. This information is found in the disk directory. Once this information has been obtained, BASIC will initialize buffer areas in memory.

After a file has been opened, the operating system will read the first sector of data. This data is passed to the memory buffer that was set up when the file was opened. BASIC may then read this data from the buffer area. After BASIC has used all the data in a sector, the operating system automatically reads the next sector into the buffer area.

When BASIC writes data to an open file, the data is first written to that file's memory buffer. Data is not actually written to the diskette until the buffer has become full. When the buffer is full, the operating system places the contents of the buffer in the correct sector of the disk.

When a file is no longer in use, it should be closed. This is especially important whenever data has been written to a file. When a file is closed, any data remaining in the buffer will be written to diskette. This occurs even if the memory buffer is not full. Then, the operating system adds the necessary directory information for that file to the diskette.

The BASIC statement, OPEN, has the following configuration:

OPEN *#filenumber, aux1, aux2, "filespec"*

filenumber is an integer that will refer to a file while it is open. Although necessary in the OPEN statement, the *filenumber* is a matter of convenience. Specifically, it would be much easier to refer to a file as #3 than by "D1:CHLAM.DAT" throughout a program. The OPEN statement assigns an integer to the file specification so that file access is simplified. Incidentally, the file specification is indicated by *filespec*.

filenumber must be in the range between 0 and 7. 0 is always reserved for the editor, while 6 is used for graphics. 7 is used to save and load programs, as well as with the LPRINT statement. As a result, only 1 through 5 are available for use with BASIC programs. 6 and 7 are available only on a limited basis. 6 is available if no graphics are used. 7 is available unless programs will be loaded or saved, or if an LPRINT statement will be executed.

aux1 and *aux2* are called auxillary parameters. Generally, *aux1* specifies the direction of data flow between the file and the computer. *aux1* is also known as the **mode**. *aux2* is a parameter that is specific to the device. For example, an 83 in *aux2* causes the Atari 820 Printer to print sideways.

Both the cassette unit and the disk drive may be opened for input (*aux1* = 4) or output (*aux1* = 8). In the **input mode** data can only be read from the file. Data cannot be written to it. If an attempt is made to open a file for input that does not already exist, a "file not found" error will occur (ERROR- 170).

The **output mode** always causes a new file to be created. If a file already exists with the same filename as that specified in the OPEN statement, existing data in that file will be erased. Data will be written to that file from its beginning point.

Besides the standard input and output modes, the disk drive supports the following advanced modes: directory (*aux1* = 6), append (*aux1* = 9), and update (aux1=12).

The **directory mode** is similar to the input mode in that nothing may be output through this mode. The difference between these two modes is that the directory mode can only use the disk directory as the input file. The following program lists all entries in the directory of drive #1 that have the extention UTL:

```
10 DIM A$(20)
20 OPEN #3,6,0,"D:*.UTL"
30 INPUT #3,A$
40 PRINT A$
50 GOTO 30
RUN
   COPY      UTL 005
   DUPDISK   UTL 004
   INIT      UTL 006
   CONVERT   UTL 005
   HELP      UTL 002
039 FREE BLOCKS

ERROR-  136 AT LINE 30
```

The ERROR-136 is an "End of file" error. It indicates that the program did not know when to stop reading data from the directory. Techniques for avoiding such errors will be discussed later in this chapter.

The file specification in the OPEN statement was "D:*.UTL". The asterisk "*" is a wildcard that may represent any string of characters. When the directory file is used as an input file, any filenames matching the file specification in the OPEN statement will be used. For example, if the file specification had been changed from "D:*.UTL" to "D:*.*", the entire directory would have been listed. The use of wild cards in *filespec* is completely analogous to their usage with the DOS command option, Files. Please refer to chapter 7 for an explanation of wildcards.

The **append mode** is specified when data is to be added to the end of an existing file. If the file to be opened for append already exists, new data will be written to the end of that file. However, if that file does not exist, a "file not found" error will occur (ERROR-170). No input may be done through a file opened for append.

The **update mode** is a combination of the input and output modes. The OPEN statement will set the **file pointer** to the beginning of the file. Any read or write operation will then advance the file pointer. The file pointer is a value stored in memory indicating the position of current data access within the file. The file pointer may not be moved past the end of the existing file. Therefore, although the contents of a file may be updated, the length of the file may not be changed in this mode.

Like the append and input modes, if the file specified in the OPEN statement does not yet exist, an ERROR-170 will occur.

The following are examples of the use of the OPEN statement:

```
OPEN#3,4,0,"K:"
OPEN#1,8,0,"C:"
```

The first example opens the keyboard as #3 for input. The second example opens the cassette unit for output using long interrecord gaps. If *aux2* had equalled 255, short gaps would have been specified.

Although seven filenumbers are available, the total number of disk files has a limit. The default limit with one drive is four simultaneously opened files.

It is good programming practice to close a file, once the program has finished accessing it. The BASIC statement, CLOSE, is used with the following configuration:

CLOSE *#filenumber*

After a file has been closed, its *filenumber* may be assigned to another file using an appropriate OPEN statement.

Writing to a Sequential File

Once a sequential file has been opened, either of the following statements can be used to output data.

```
PRINT#
PUT#
```

PRINT# functions almost exactly as does PRINT. The difference lies in the fact that PRINT# requires that a file number be specified. Data is written to that file rather than to the display. An example of PRINT# is given below:

```
10 OPEN #2,8,0,"D:FILE.DAT"
20 A = 27.932
30 PRINT #2; A, "J. C."
40 PRINT #2; "REBEL"
```

The following will be saved by the PRINT# statements in lines 30 and 40:

```
27.932       J.C. ←
REBEL   ←────────────   CR & LF characters
```

This output can be verified by substituting "S:" for "D:FILE.DAT" in line 10. This change causes the output to appear on the screen.

The PUT# statement is used to send one byte of data to a particular device. The following two statements are equivalent:

```
PUT #3, 65
PRINT #3; CHR$ (65);
```

Notice that the equivalent PRINT# statement has a trailing semicolon to suppress the carriage return which is generated by an ordinary PRINT# statement. PUT# is generally used as a shorthand for the equivalent PRINT# statement. It is most useful for single byte data transfers. The following program outputs the characters that correspond to the ASCII codes 0 through 100. The output device is the screen.

```
10 OPEN #3,8,0,"S:"
20 FOR I=0 TO 100
30 PUT #3,I
40 NEXT I
```

Reading from a Sequential File

The following commands are used in Atari BASIC to input data from a sequential file:

```
INPUT#
GET#
```

INPUT# functions with sequential files much like INPUT does with the keyboard. INPUT# will read the data at the current position in the sequential file and assign that data to the variable indicated as its argument. The data and variable must be of the same type. If they are not, an error condition will result.

When INPUT# is reading numeric data, any leading blanks will be ignored. As is the case with INPUT, CR/LF characters and commas may be used as delimiters. Any non-numeric characters, excluding leading spaces, will result in an error.

When INPUT# is reading string data, all characters up to the next carriage return will be assigned to the string. This assumes that the string variable has been dimensioned sufficiently large to accommodate the data. If the string variable is not large enough, as much as will fit will be placed in the variable. Commas, spaces and semi-colons will be treated as data, not as delimiters.

```
100 DIM A$(20),B$(20)
110 OPEN #3,8,0,"D:DATA"
120 PRINT #3;"John"
130 PRINT #3;"Smith"
140 CLOSE #3
150 OPEN #3,4,0,"D:DATA"
160 INPUT #3;A$
170 INPUT #3;B$
180 PRINT A$,B$
```

In the preceding example, the data was first written to the disk file, then retrieved using two INPUT# statements. Lines 160 and 170 could have been combined into the following line:

```
160 INPUT #3;A$,B$
```

The GET# statement is used to retrieve one byte of data from a device. GET# is not limited by the carriage return delimiter because it always fetches one byte, regardless of that byte's value.

The following program will output the data contained in "D:DATA" to the screen. The infinite loop in lines 120-140 will continually reexecute until the "End of file" is reached. Here, an error will occur. The technique for avoiding this error is discussed in the next section.

```
100 OPEN #1,4,0,"D:DATA"
110 OPEN #2,8,0,"S:"
120 GET #1,X
130 PUT #2,X
140 GOTO 120
```

AVOIDING EOF ERRORS

Atari BASIC does not contain an explicit EOF function — one that indicates whether the end of file has been reached. Therefore, this error must be side-stepped using the TRAP statement. If the last example is edited as follows, the error will be avoided:

```
90 TRAP 150
100 OPEN #1,4,0,"D:DATA"
110 OPEN #2,8,0,"S:"
120 GET #1,X
130 PUT #2,X
140 GOTO 120
150 REM ***** ERROR ROUTINE *****
160 ERR=PEEK(195)
170 IF ERR=136 THEN END
180 PRINT "ERROR—";ERR
```

The TRAP statement at line 90 branches program control to the subroutine at line 150 in the event of an error. However, TRAP does not discriminate between errors — any error will cause the execution of the subroutine. The routine must verify that the error was indeed caused by the "End of file" condition.

Recall that PEEK(195) returns the error number. This number is compared to the error code for the EOF (136). If the error was an EOF error, the program ends normally. Otherwise, the error code is printed. To demonstrate this, power-down the disk drive before executing the program. "ERROR—138" (Device timeout) should appear on the screen.

Random Files

Generally, files are created sequentially in Atari BASIC. BASIC does not include commands specifically designed to create random access files. For example, many versions of BASIC include the command FIELD. This command insures that each record of the file occupies the same amount of disk space. Recall that records with equivalent lengths are necessary for random access.

The creation of a random access file may be simulated by using the PRINT# and GET# statements. The programmer is then left with the responsibility to write records with equal lengths. Therefore, true random access is rarely implemented in Atari BASIC unless frequent and/or fast file updates are required.

NOTE & POINT

Random access in Atari BASIC is usually limited to a pseudo-random access accomplished by the commands NOTE and POINT. Files used with these commands are generally written sequentially and read randomly. To eliminate the delays of sequential access, the NOTE command is used to remember the location of each record in the file. Later, the POINT command may be used to place the file pointer at the beginning of any record. NOTE and POINT are used with the following configurations:

```
NOTE #filenumber, variable1, variable2
POINT #filenumber, variable1, variable2
```

The significance of *variable1* and *variable2* are the same for both version 2.0S and 2.5 DOS. *Variable1* indicates the absolute sector number (1-719), while *variable2* indicates the character number within the sector (0-125). Notice that care must be taken when using POINT. The file pointer could easily be moved to a place on the disk that does not contain the correct file. The specified sector is not verified as part of the file until a read or write operation is performed.

Here, one of the following errors may occur:

ERROR-170	attempted READ outside file
ERROR-171	attempted WRITE outside file

Neither NOTE nor POINT will operate correctly with version 1.0 of the disk operating system. Therefore, random access may not be accomplished using DOS 1.0.

EXTENDED INPUT AND OUTPUT COMMANDS

Atari BASIC includes the extended input-output command — XIO. The XIO command may be used in a plethora of applications; many of these are related to disk access. XIO is used with the following configuration:

XIO *command*, *#filenumber*, *aux1*, *aux2*, aux3

command is an integer that selects the desired I/O operation. Table 5-2 lists the operations discussed in this section. Chapter 8, "BASIC Reference Guide", lists the XIO commands in their entirety.

filenumber must be the same as the one used when the file was opened. *aux1* specifies the direction of data flow — input (*aux1* = 4) or output (*aux1* = 8). *aux2* is not used and should be set to 0.

Table 5.2. Extended input-output commands

command	data direction	operation
5	input	read line
7	input	read record (255 characters)
9	output	write line
11	output	write record (255 characters)

Generally, *aux3* is the string variable through which input or output is done. For output operations, *aux3* may be a string constant. If *aux3* is a variable, it must be dimensioned before the XIO command is executed.

XIO 5 and XIO 9 are similar to INPUT# and PRINT#, respectively. The first example program in this section will illustrate the difference between INPUT# and XIO 5, while the second program will differentiate PRINT# and XIO 9.

```
10 DIM A$(9), B$(9), C$(9)
20 POKE 201,11
100 OPEN #3,8,0,"D:DATA"
110 PRINT #3;"MERRY CHRISTMAS, EVERYONE!"
120 PRINT #3; "HO, HO, HO!"
130 CLOSE #3
200 OPEN #3,4,0,"D:DATA"
210 INPUT #3,A$
220 INPUT #3,B$
230 CLOSE #3
240 PRINT A$,B$
300 OPEN #3,4,0,"D:DATA"
310 XIO 5,#3,4,0,A$
320 CLOSE #3
330 PRINT A$,B$,C$
340 C$(9)="#"
350 PRINT A$,B$,C$
RUN
MERRY CHR  HO, HO,  H
MERRY CHR  HSTMAS, E
MERRY CHR  HSTMAS, E  VERYONE!#
```

Both XIO 5 and INPUT# will read the input file to an "end of line" character. INPUT# will then discard any data that cannot fit into the specific string variable; however, XIO 5 will continue to store this data in successive memory locations. These locations are generally assigned to another string variable. Therefore, a single XIO statement may be used to load several string variables.

XIO 5 does have a few quirks that must be understood to effectively implement this command. Although the INPUT command does not store the EOL character, XIO 5 does. The INPUT command will change the length of the string variable to the number of characters input, where XIO 5 will not adjust the length of the string. (Notice the final output line of the example program.) The XIO 5 command can load more than one variable; however, the first memory location after the current length of the specified variable will not be changed. (This fact is illustrated in the second output line of the example program.)

Incidentally, line 20 of the program merely adjusts the tabulation width of the display. All successive commas in PRINT statements will cause the output to begin in the next 11 character print zone. This was done to facilitate a more pleasing display and has nothing to do with the concept of the program.

The XIO 9 command will write characters from a specified string. The string will be output until an EOL is encountered. If the string does not contain an EOL character, one will appended to the output. XIO 9 is similar to PRINT# except that PRINT# will write the entire string regardless of contents. The ASCII value of the EOL character is 155 decimal.

```
10 DIM A$(20)
20 A$="HAPPY EASTER BUNNY"
100 OPEN #3,8,0,"E:"
110 PRINT A$
115 PRINT
120 A$(13,13)=CHR$(155)
130 PRINT #3;A$
135 PRINT
140 XIO 9,#3,8,0,A$
150 CLOSE #3
```

When the previous program is executed, the following will be output:

```
HAPPY EASTER BUNNY

HAPPY EASTER
BUNNY

HAPPY EASTER
```

The XIO 7 and XIO 11 commands are used to read and write records of 255 characters, respectively. Because they transfer a fixed number of bytes, these commands are generally not useful for random string storage. However, the commands are useful for array storage because the contents of the transferred bytes have no effect on the operation of the commands.

Like the XIO 5 commands, after XIO 7 fills the specified string to its current length, the next byte read isn't stored in memory. When XIO 11 is executed, the specified string will be output to its current length. Then, regardless of the next byte's value, an EOL will be output. After the EOL, the balance of the 255 characters will be written.

The following statements will store the array, ELAINE, as well as the strings — MARKER$ and DUMMY$.

```
10 OPEN #1,8,0,"D:ARRAY"
20 DIM MARKER$(1),ELAINE(5,6),DUMMY$(2)
30 XIO 11,#1,8,0,MARKER$
40 CLOSE #1
```

MARKER$ indicates where the array, ELAINE, can be found in memory. DUMMY$ occupies the rest of the 255 bytes stored by XIO 11. (An array requires 6 bytes per element — 6 bytes * 6 * 7 = 252) The following statements will recover the array:

```
10 OPEN #1,4,0,"D:ARRAY"
20 DIM MARKER$(1), ELAINE(5,6),DUMMY$(2)
30 XIO 7,#1,4,0,MARKER$
40 CLOSE #1
```

File Commands

Atari BASIC includes five commands designed to perform file handling operations while the BASIC interpreter is active. These include SAVE, LOAD, RUN, LIST, and ENTER. The extended input-output command, XIO, is also available. XIO can be used to erase, rename, protect or unprotect a file and to format a disk.

SAVE

SAVE generally is used to store a program on a cassette or disk file. Before storing, a program is encoded into a tokenized form. This form allows the program to consume less disk space and to load more quickly. SAVE is used with the following configuration:

SAVE "*filespec*"

filespec is composed of a device name and a filename. The device name specifies where to save the program, while the filename specifies what to call the program.

```
SAVE "D2:GLADYS"
```

The previous example would save the program in RAM in a file named "GLADYS" on disk drive #2.

LOAD

The LOAD command is generally used to load a program file into memory from cassette or diskette. LOAD recognizes only the tokenized format used by the SAVE command.

LOAD "*filespec*"

LOAD erases any program lines and variables in memory before the specified program is loaded.

RUN

The RUN command causes the computer to both load and execute the designated file specification. This file must have been stored in the tokenized format.

RUN "*filespec*"

When used as a program line, RUN facilitates program concatenation. A complex program may overrun the Atari's memory limitations. When this is the case, the program can be separated into two or more self-sufficient parts. If a portion of the program is needed that is not currently in memory, it can be loaded to continue the work that the previous portion had accomplished.

The following program will display the disk directory. Any tokenized BASIC program on the disk may be executed at the touch of a key. This is a convenient program to have on every diskette containing BASIC programs.

```
100 DIM FILE$(20),EXTENTION$(3)
110 OPEN #1,4,0,"K:"
120 OPEN #2,5,0,"E:"
130 OPEN #3,6,0,"D:*.*"
200 REM
210 REM READ DIRECTORY
220 REM
230 TRAP 300
240 CHAR=65
250 INPUT #3;FILE$
260 PRINT CHR$(CHAR);" ";FILE$
270 CHAR=CHAR+1
280 IF CHAR<86 THEN 250
300 REM
310 REM WHICH FILE????
320 REM
330 TRAP 600
340 GET #1,PROGNUM
350 POSITION 4,PROGNUM-65
360 INPUT #2;FILE$
400 REM
410 REM RUN PROGRAM
420 REM
430 FILE$(1,2)="D:"
440 EXTENTION$=FILE$(11,13)
450 POS=1
460 IF FILE$(POS,POS)<>" "AND POS<11 THEN POS=POS+1:GOTO 460
470 FILE$(POS)="."
480 FILE$(POS+1)=EXTENTION$
490 POSITION 5,22
500 PRINT "LOADING........";FILE$;"       "
510 RUN FILE$
600 REM
610 REM ERROR ROUTINE
620 REM
630 POSITION 5,22
640 PRINT "SELECT AGAIN    ERROR—";PEEK(195);"      "
650 GOTO 300
```

Lines 100-130 are the initialization procedures. Here, the keyboard and screen editor will be opened for input. The screen editor will not be opened in the conventional manner (*aux1* = 4) because the editor has two input modes. The standard mode requires that the RETURN key be pressed to input data, while the force-read mode (*aux1* = 5) eliminates this requirement. In the forced-read mode, the computer will generate a return regardless of operator interaction.

Lines 200-280 will list the disk directory. Each entry will be prefixed with a letter to more easily indicate a selection. Lines 300-360 will wait for an appropriate keypress, then will move the name of the selected file from the screen to the variable FILE$.

Lines 400-480 will manipulate FILE$ until its form matches that required by a RUN *"filespec"* statement. Then, line 510 will execute the desired file.

LIST

The LIST statement is used with the following configuration to display or record programs found in the computer's memory:

LIST *"filespec", linenumber1, linenumber2*

The LIST statement can be used to save a program, or part of a program, on disk or cassette. It operates in a manner similar to the SAVE command. The major difference is that any program stored using LIST is not placed into a tokenized form. Therefore, programs stored with LIST may not be retrieved by either LOAD or RUN. ENTER is the only BASIC statement that can recover a program saved by LIST.

Untokenized programs are stored in an ASCII format, and therefore require more disk space than do equivalent tokenized programs. Also, untokenized programs load more slowly. However, ASCII formatted programs may be merged, whereas tokenized programs may not.

The optional parameters, *linenumber1* and *linenumber2* specify the range of program lines to be saved by the command. If these are omitted, the entire RAM-resident program will be stored.

filespec indicates the device and filename used to save the program. If *filespec* is omitted, the screen editor is used by default.

ENTER

The ENTER statement loads the specified program file into memory and merges it with the existing RAM-resident program lines.

ENTER *"filespec"*

For a program to be loaded with ENTER, it must have been saved in ASCII format using the LIST command. If the file being loaded contains a program line with the same line number as one of the program lines already present in memory, the program line being loaded will replace that line.

Suppose that two parts of a program have been developed separately. Now they must be combined, so that they may be loaded with a single command. These parts are stored on diskette using the names —"PROGA" and "PROGB", respectively. The following sequence of commands will combine the two programs and store the result with the filename "FINAL":

```
LOAD "D:PROGA"
LIST "D:PROGA"
LOAD "D:PROGB"
ENTER "D:PROGA"
SAVE "D:FINAL"
```

The first two lines will place "PROGA" into ASCII form. Line three loads "PROGB", erasing "PROGA" from memory. Line four merges the two programs. Finally, line five saves the result.

ERASE (XIO 33)

"Erase" is used with the following configuration to delete the disk file indicated by *filespec*:

XIO 33,#7,0,0,*"filespec"*

RENAME (XIO 32)

"Rename" is used with the following configuration to change a filename. The filename included in *filespec* will be changed to that specified in *newname*.

XIO 32,#7,0,0,"*filespec,newname*"

As an example, the following command will affect the file named "HAUPT" on drive #4. The filename "HAUPT" will be replaced with its *newname*, "KLING".

```
XIO 32,#7,0,0,"D4:HAUPT,KLING"
```

PROTECT (XIO 35)

A protected file may not be erased or replaced by a file of the same name. Also, nothing may be appended to a protected file. "Protect" uses the following configuration to mark a directory entry as a permanent file.

XIO 35,#7,0,0,"*filespec*"

UNPROTECT (XIO 36)

"Unprotect" uses the following configuration to release a file from its protected state:

XIO 36,#7,0,0,"*filespec*"

Suppose that a BASIC program is stored under the protected filename, "ANDY", and that the program has just been edited. Now, the revised program must be stored again using the old filename. The following sequence of commands will unprotect, re-save, and then re-protect the program:

```
XIO 36,#7,0,0,"D:ANDY"
SAVE "D:ANDY"
XIO 35,#7,0,0,"D:ANDY"
```

FORMAT (XIO 253 & XIO 254)

Atari manufactures two disk drives — the 810 and the 1050. The 810 can only format in single-density (90K); whereas, the 1050 can format in both single-density and dual-density (130K). The appropriate format command may be determined by the desired format density.

XIO 253,#7,33,*aux2*,"D*drivenum*:"
XIO 254,#7,0,0,"D*drivenum*:"

XIO 254 can only be used to format a single-density disk. XIO 253 may be used for either single-density or dual-density formatting. *drivenum* indicates which drive to format. Generally, this number ranges from 1 to 4.

When using XIO 253, *aux2* determines the format density. *aux2* = 87 indicates single-density. Likewise, *aux2* = 127 specifies dual-density. If *aux2* is assigned any other value, a non-standard disk format will result.

6

BASIC GRAPHICS & SOUND

Introduction

The Atari 130XE has 15 graphics modes encompassing 256 colors. This gives the 130XE some of the best color graphics capabilities available on a home computer.

Besides the many graphics modes available on the 130XE, the computer has sophisticated BASIC supported, sound capabilities, including sound effects and four channel music. These are generated by POKEY —Atari's custom input, output, and sound IC chip. The use of POKEY to generate complex sounds will be discussed later in this chapter.

The Atari's forte is obviously its graphics and sound. This chapter is designed to familiarize the user with the graphics capabilities available in Atari BASIC. But, to paraphrase Shakespeare, "There are more things in heaven and earth than are dreamt of in your BASIC." In general, Atari BASIC is not equipped to support the graphics capability of the hardware. Graphics features not well supported in BASIC will be discussed in various appendices.

The Graphics Modes

Atari BASIC supports fifteen graphics modes, all of which are color capable. However, the maximum number of concurrently displayable colors is limited by the selected mode. Of the fifteen modes, eleven use pixel graphics, while the other four use character graphics.

PIXELS

In pixel graphics, the display can be divided into a grid. Every point on the screen can be uniquely indentified by its row and column numbers. For example, the screen element at column 15 and row 9 can be specified by the ordered pair (15,9). Notice that the column number is listed first. Each specific screen element is called a pixel.

A pixel can be assigned a single color. Pictures formed using pixel graphics are generated by assigning appropriate colors to a number of pixels. Since there is no space between pixels, assigning the same color to adjacent pixels will cause that portion of the screen to appear as a solid color.

In low resolution graphics, the display can be divided into a grid of 24 rows and 40 columns. The farthest left column of the screen has been defined as column 0. The farthest right column has been defined as column 39. In a similar manner, the row numbers extend from 0 (top) to 23 (bottom). This arrangement may seem upside down to a person familiar with a cartesian coordinate system.

The remainder of the pixel graphics modes can likewise be divided into grids of 80 x 48, 160 x 96, 160 x 192, or 320 x 192. The selected graphics mode determines the screen resolution. In the case of high resolution graphics, the column numbers now extend from 0 (left) to 319 (right), while the rows are numbered from 0 (top) to 191 (bottom).

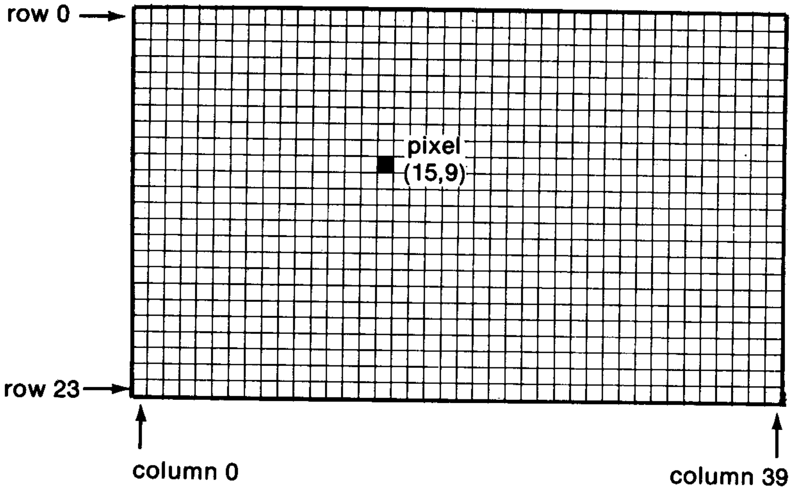


Figure 6.1. Low resolution pixels

CHARACTER GRAPHICS

Character graphics differ from pixel graphics in that objects drawn with character graphics must be predefined. For example, enter the following two lines:

```
GRAPHICS 2
PRINT#6,"A"
```

The large capital A, drawn by the preceding lines, is a character. Nowhere in these two lines is the computer told what an uppercase-A looks like. None the less, an A was drawn. This character had been predefined in ROM. Incidentally, the text mode is a type of character graphics mode.

The programmer has the ability to define his own characters. For example, a rocket ship or a man could be defined as a character. Changing the definition of a character will immediately change the appearance of the character on the screen.

```
POKE 756,226
```

The previous statement will change the character set causing the displayed characters to change in appearance. A dynamic character set is an effective way to animate screen images. Appendix F discusses the creation of a custom character set.

SELECTING A GRAPHICS MODE

The GRAPHICS statement allows the programmer to select between the fifteen graphics modes and the text mode. GRAPHICS is used with the following configuration:

GRAPHICS *argument*

argument indicates the display mode. GRAPHICS 0 selects the text mode. GRAPHICS 1 through GRAPHICS 15 selects the graphics modes. This is summarized in table 6.1. Only GRAPHICS 0 through GRAPHICS 11 are supported on earlier Ataris.

The GRAPHICS statement generally clears the screen display upon execution. Adding 32 to *argument* suppresses this feature.

Likewise, adding 16 to *argument* suppresses the text window. In modes 1-8 and 12-15, four lines of text known as the text window are located beneath the graphics display. To accommodate the text window, the screen resolution must be reduced. For example, a high resolution screen without a text window has a resolution of 320 x 192 pixels; however, with a text window the resolution of the screen is reduced to 320 x 160 pixels. Table 6.1 lists both full-screen and split-screen resolutions.

```
GRAPHICS 3+16
GRAPHICS 7+32
```

Of the preceding statements, the first will configure the Atari to display a full-screen of graphics mode 3. The screen will be cleared upon its execution. The second will configure the Atari to a mode 7 screen with a text window; however, the screen will not be cleared by this command.

Table 6.1. Atari graphics modes

Graphics Mode	Mode Type	Resolution Full Screen	Resolution Split Screen	Number Of Colors
0	Text	40 x 24	—	1*
1	Character	20 x 24	20 x 20	5
2	Character	20 x 12	20 x 10	5
3	Pixel	40 x 24	40 x 20	4
4	Pixel	80 x 48	80 x 40	2
5	Pixel	80 x 48	80 x 40	4
6	Pixel	160 x 96	160 x 80	2
7	Pixel	160 x 96	160 x 80	4
8	Pixel	320 x 192	320 x 160	1*
9	Pixel	80 x 192	—	1**
10	Pixel	80 x 192	—	9
11	Pixel	80 x 192	—	16***
12	Character	40 x 24	40 x 20	5
13	Character	40 x 12	40 x 10	5
14	Pixel	160 x 192	160 x 160	2
15	Pixel	160 x 192	160 x 160	4

Color Registers

The Atari 130XE can display 16 different hues in 16 luminances (or shades) for a total of 256 displayable colors. Although the number of concurrently displayable colors is generally limited to 2 or 4, these colors may be any from the pallette of 256.

The screen color registers are memory locations within the Atari that determine the foreground, background, and border colors. The color

* 1 Hue; 2 Luminances
** 1 Hue; 16 Luminances
*** 16 Hues; 1 Luminances

registers record both the hue and luminance with which to display the color. The Atari's operating system uses the following RAM addresses to store the contents of the five registers:

Address	Color Register	Default Color
708	0	ORANGE
709	1	GREEN
710	2	BLUE
711	3	RED
712	4	BLACK

The default color values for the five color registers can be changed with the SETCOLOR command. SETCOLOR is used with the following configuration:

SETCOLOR *register, hue, luminance*

The first argument of SETCOLOR indicates which register to set. The second argument selects the hue itself and can range from 0 to 15. Table 6.2 lists the available hues and their corresponding numbers. The final argument of SETCOLOR determines the brightness of the color and can also range from 0 (darkest) to 15 (brightest).

Generally, registers 0-3 each determine a foreground color, while register 4 controls the background and border color. However, this is not always the case — for example, register 2 controls the background in mode 0 (text mode). Table 6.2 lists the color register control assignments.

As an example, the following statement, when executed in mode 0, will set the background color of the screen to black (0 = grey, 0 = darkest).

```
SETCOLOR 2,0,0
```

Table 6.2. Hue vs. hue numbers

Hue	Hue number
Gray	0
Gold	1
Orange	2
Red	3
Pink	4
Violet	5
Blue-Violet	6
Blue	7
Blue	8
Light Blue	9
Turquoise	10
Green-Blue	11
Green	12
Yellow-Green	13
Orange-Green	14
Orange	15

Commands Used with Pixel Graphics

SELECTING A COLOR REGISTER

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

COLOR *argument*

Whenever a graphics output command such as PLOT or DRAWTO is issued, the color register selected by the most recent COLOR statement is used. *argument* indirectly specifies the desired color register. Generally, COLOR 0 selects the background color (color register 4). *arguments* greater than zero select one of the foreground colors.

COLOR 0 ⟶ register 4
COLOR 1 ⟶ register 0
COLOR 2 ⟶ register 1
COLOR 3 ⟶ register 2

The previous assignments are valid in modes 3-7, 14 and 15. Modes 4, 6 and 14 are two color modes (see table 6.1); therefore, COLOR 2 and COLOR 3 will not operate correctly in these modes. The GTIA modes (9-11) use the COLOR command differently; this will be discussed in a later section.

PLOTTING

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

PLOT *column, row*

column and *row* specify the coordinate of the pixel to be illuminated. The range of these arguments is determined by the current graphics modes.

ADVANCED GRAPHICS COMMANDS

With the right combination of PLOT and elbow grease, any graphics screen can be drawn. In other words, although PLOT gets the job done, it does not accomplish it with a great deal of efficiency. BASIC includes two commands that can simplify the creation of graphics displays — DRAWTO and XIO 18.

DRAWTO is used to plot consecutive pixels. For example, the following statements will connect the pixel (10,10) to pixel (150,70) with an orange line.

```
GRAPHICS 14
COLOR 1
PLOT 10,10
DRAWTO 150,70
```

DRAWTO connects the last pixel referenced to the coordinate specified as DRAWTO's argument. The last pixel referenced can be set by either a PLOT statement or a previous DRAWTO statement. For example, the following statements draw an orange triangle:

```
GRAPHICS 6
COLOR 1
PLOT 80,20
DRAWTO 60,40
DRAWTO 85,45
DRAWTO 80,20
```

The second advanced graphics command is XIO 18, commonly known as "fill". This command will paint a bounded section of the screen with a user determined color. The correct syntax of this command is as follows:

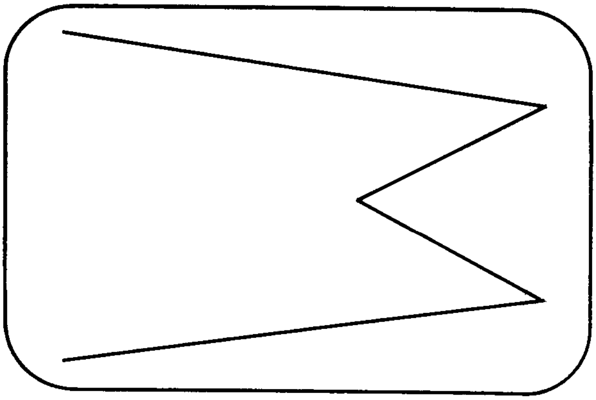
```
XIO 18, #6, 0, 0, "S:"
```

"18" signifies the fill operation. "#6" is the filenumber used with the graphics modes. Finally, "S:" indicates the screen device. (All graphics modes output through the screen device.)

Painting an area involves more than including the fill command in a program. The boundaries of the area must first be defined. The first step is to draw the right edge of the figure. This edge can be as complicated as desired and drawn in any foreground color.

```
10 GRAPHICS 15
20 COLOR 2
30 PLOT 10,150
40 DRAWTO 150,140
50 DRAWTO 80,80
60 DRAWTO 150,20
70 DRAWTO 10,10
```

The preceding program will cause the following graphics output:

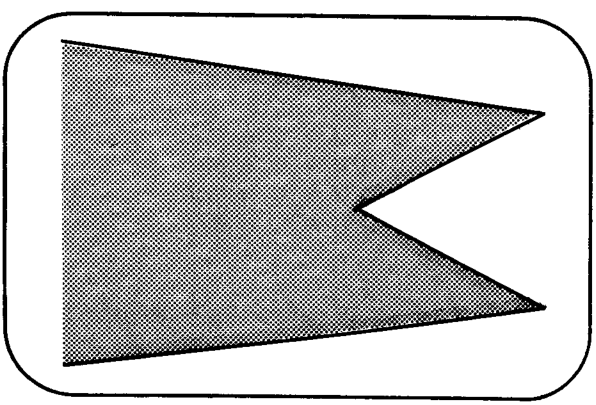


The next step is to execute a POSITION statement to the screen coordinate of the lower left corner of the figure. In our case, this is (10,150). The color with which to fill the area should then be poked into location 765.

```
80 POSITION 10,150
90 POKE 765,1
```

Finally, the XIO 18 command should be executed. The enclosed area will then be painted.

```
100 XIO 18, #6, 0, 0, "S:"
RUN
```



GTIA GRAPHICS

The color selection schemes in modes 9, 10, and 11 differ from that in the other pixel graphics modes. These color selection schemes are outlined in the BASIC Reference chapter under the COLOR command.

Mode 9 is used extensively for creating 3-dimensional images. It can display 16 luminances of any single hue. The hue is determined by the value stored in color register #4. SETCOLOR 4 is the easiest way to set this value. The luminance argument of SETCOLOR 4 should be set to 0 when using mode 9.

```
100 GRAPHICS 9
110 SETCOLOR 4,0,0
120 FOR I=0 TO 191
130 COLOR INT(I/3)
140 PLOT 0,I:DRAWTO 79,I
150 NEXT I
200 FOR X=0 TO 64 STEP 16
210 FOR Y=0 TO 144 STEP 48
220 GOSUB 300
230 NEXT Y
240 NEXT X
250 GOTO 250
300 FOR I=0 TO 15
310 COLOR I
320 PLOT X+I,Y:DRAWTO X+I,Y+I*3
330 NEXT I
340 RETURN
```

Mode 11 allows a sixteen hue graphics display. Here, color register #4 determines the luminance of the screen, while the COLOR statement determines the hue. The hue argument of SETCOLOR 4 should be set to 0 for proper results.

Commands Used with Character Graphics

Modes 1 and 2 can be used to display enlarged text. The items available for display can be chosen from one of three character sets. The standard character set consists of the uppercase letters, digits, and punctuation symbols. The alternate character set consists of the lowercase letters and special graphics characters. The extended characters consist of a number of international symbols. (ex. ä)

The standard character set will be active whenever the Atari is powered-on, when the RESET key is pressed, or when a GRAPHICS statement is executed. Location 756 determines the active character set.

POKE 756,206	extended
POKE 756,224	standard
POKE 756,226	alternate

In mode 1, the characters are printed at the same height as those in the text mode (0); however, they are printed at twice the width. In mode 2, the characters are printed at twice the height and width of those in the text mode.

When a GRAPHICS statement is issued, filenumber 6 is opened to the screen device (S:). Therefore, a PRINT#6 or PUT#6 will cause data to be output to the graphics display.

Five different default colors are available in graphics modes 1 and 2. These correspond to color registers 0 through 4. Color register 4 controls the background and border colors. The default color is hue = 0; luminance = 0. This sets the background and border colors to black. SETCOLOR 4,0,4 will set the border and background colors to grey in graphics modes 1 and 2. SETCOLOR 4,2,4 will set the background and border colors to orange.

The BASIC reference chapter gives the procedure for determining which color register is used to draw a character in these modes. This information is listed under the PRINT# and PUT# statements.

SOUND

In Atari BASIC, the SOUND statement is used to output music or noise 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 the four independent voices. These are numbered 0 to 3.

pitch sets the frequency of the sound produced by the SOUND statement. *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 from 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, 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, and 15) 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 RESET key is pressed, sound will be turned off. However, if the BREAK key is pressed, sound will not be turned off.

Writing A Game Program

In this section, the game, BARACADE, will be designed. The object of the game is to avoid the obstacles, your trail, and your opponent's trail. The game will be written in BASIC so that it may easily be modified. If the reader does not wish to follow the step by step designing of BARACADE, he may page through the chapter. All program lines may easily be distinguished from the rest of the text. To play BARACADE, merely enter every line belonging to the program.

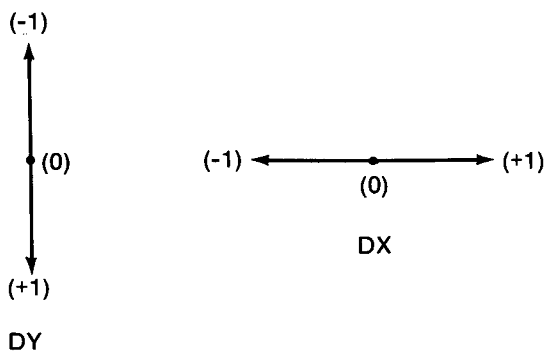
The first step in designing BARACADE is to program the computer to draw a trail. The following statements accomplish this:

```
100 DIM X(1),Y(1),DX(1),DY(1),SCORE(1)
130 X(0)=15:Y(0)=11
140 DX(0)=1:DY(0)=0
170 GRAPHICS 19:REM no text window
230 S=STICK(I)
250 DX=(S=7)-(S=11)
260 DY=(S=13)-(S=14)
```

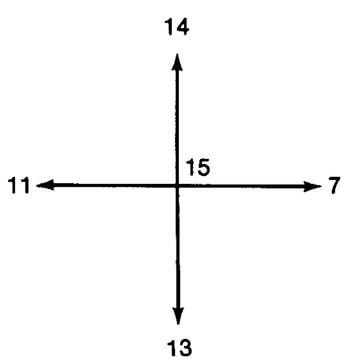
program continued on next page

```
280 IF DX OR DY THEN DX(I)=DX:DY(I)=DY
290 X(I)=X(I)+DX(I)
300 Y(I)=Y(I)+DY(I)
330 COLOR I+1
340 PLOT X(I),Y(I)
380 GOTO 230
```

Line 130 sets the initial position at screen location (15,11). The DX and DY in line 140 are the direction variables. Initially, the direction of movement is set to the right (DX = 1; DY = 0).



Lines 230 through 380 set up a loop that monitors joystick #1 and then act accordingly. Since I = 0 everywhere in the program, the variable S is assigned the value of STICK(0). The value returned corresponds to the position of joystick #1. STICK(1) will later be used with joystick #2.



Lines 250-280 recalculate the direction variables based on the joystick position. The numeric variables DX and DY receive the temporary result of STICK. If either DX or DY is non zero, their values are stored in the array variables DX(I) and DY(I). Therefore, the array variables are only affected when the joystick is being manipulated.

The current position variables are recalculated in line 290 and 300, while line 340 plots the new position. Executing the program is the best way to understand how it operates.

The program has been written so as to simplify the addition of a second player. Array variables were used so that the same loop can control both players. By including the following three lines in the program, a second player can enjoy BARACADE:

```
150 X(1)=25:Y(1)=11
160 DX(1)=-1:DY(1)=0
350 I= NOT I
```

Line 150 and 160 set the initial position and direction of the second player. Line 350 toggles between selecting player #1 (I=0) and player #2 (I=1).

Recall, from our description of BARACADE, that the purpose of the game is for the player to avoid colliding with any obstacles. The LOCATE statement will be used to check for collisions. If the following lines are added to the program, collisions will be detected:

```
310 LOCATE X(I),Y(I),COLLISION
320 IF COLLISION THEN 440
```

The program has not yet been completed. When it is run, an error will occur after every collision. This is because the computer has not yet been instructed what to do upon collision. Let's tell it by adding the following lines to the program:

```
440 GRAPHICS 35
450 POKE 752,1
460 IF I<>1 THEN PRINT "GREEN";
470 IF I<>0 THEN PRINT "ORANGE";
480 PRINT " WINS"
540 IF STRIG(0) OR STRIG(1) THEN 540
550 GOTO 130
```

Line 460 and 470 determine the winning player. At this point in the program, I is equal to the number of the player who just collided with something. Line 540 delays the computer until both players are ready for another game. Pressing the button on the joystick indicates that a player is ready.

A playing field can be added by using the following lines:

```
180 COLOR 3
190 PLOT 0,0
200 DRAWTO 39,0:DRAWTO 39,23
210 DRAWTO 0,23:DRAWTO 0,0
```

To keep track of the score, add the following lines to the BARACADE program:

```
110 PRINT "■TO BEGIN PRESS JOYSTICK TRIGGER"
120 GOTO 630
220 BLOCKS=0
370 BLOCKS=BLOCKS+1
480 PRINT " WINS ";BLOCKS;" BLOCKS"
490 SCORE(1-I)=SCORE(1-I)+BLOCKS
500 PRINT ,,"ORANGE = ";SCORE(0)
510 PRINT ,," GREEN = ";SCORE(1)
520 IF SCORE(0)>999 THEN 560
530 IF SCORE(1)>999 THEN 560
560 GRAPHICS 18
570 IF I<>0 THEN PRINT #6;"ORANGE";
580 IF I<>1 THEN PRINT #6;"GREEN";
590 PRINT #6;" WINS"
600 POSITION 0,3
```

program continued on next page

```
610 PRINT #6;"FINAL SCORE"
620 PRINT #6;SCORE(0),SCORE(1)
630 SCORE(0)=0:SCORE(1)=0
640 GOTO 540
```

When the preceding lines are added to the BARACADE program, a single BARACADE match will consist of a number of BARACADE games. The first player to claim 1000 screen blocks will be declared the winner. Lines 520 and 530 determine the point total needed for victory.

Arcade sound may be added to BARACADE by adding the following lines. The sound of an explosion is simulated in line 390. The loop at lines 400-420 cause the losing player to flash, as if exploding.

```
240 SOUND 0,0,0,0
320 IF COLLISION THEN 390
360 SOUND 0,I*50+10,10,8
390 SOUND 0,100,4,15
400 FOR J=0 TO 127
410 SETCOLOR I,0,J
420 NEXT J
430 SOUND 0,0,0,0
```

The program as it stands has one minor bug. If a player tries to change direction by 180°, he will lose. This is because, as far as the computer is concerned, the player ran into himself. Although this does not detract from game play, it can be annoying. When the following line is added to the program, the bug will be corrected.

```
270 IF DX AND DX(I) OR DY AND DY(I) THEN 290
```

The ideas in this section by no means exhaust the possibilities that could be added to BARACADE. Other upgrades might include: keeping track of matches won, adding a more complex playing field, or making one player faster than the other. The only two limiting factors are execution speed and one's imagination.

7

DOS Usage

Introduction

Atari DOS, or disk operating system, is a group of programs that allows the user to manage information on diskette. The DOS programs (commands) are provided on a diskette known as the master diskette. The diskette should not be used in day to day operations — copies should be used instead. Procedures for copying the master diskette will be detailed in this chapter. DOS command usage will also be explained.

There are several versions of DOS that can be used with Atari home computers. This chapter will focus specifically on versions 2.0S and 2.5 —the two most popular Atari disk operating systems. The label on the master diskette should specify the version of DOS.

Disk Files

Both of the Atari disk operating systems store data in files. A file is a group of related information. For example, a file might consists of a list of customers or perhaps might contain the text of a standard form letter. A file could also contain a program to edit and print the form letters. The advantage of grouping information in a file is that the file can then be easily accessed by DOS.

A number of files can be stored on a single diskette. Both Atari DOS 2.0S and 2.5 allow up to 64 files per diskette. Every file stored on a specific diskette must have a unique **filename**. A filename consists of a primary filename and an optional filename extension. Examples of filenames are given below. Note that the second example does not contain a filename extension.

```
GRIM.JIM
ARNE18
PHONE.BK
DONKEYKONG.JR
```

DOS allows primary filenames of up to eight alphanumeric characters in length. Valid characters include the letters **A** through **Z** and the digits **0** through **9**.

The filename extension is an optional name that can appear after the primary filename. The filename extension begins with a period followed by one, two, or three characters. When a filename extension is included in a filename, both the primary filename and the extension must be used to identify the file.

Filename extensions are often used to indicate the type of file. Commonly used filename extensions and their corresponding meanings are listed in table 7.1.

Table 7.1. Commonly used filename extensions

Filename Extension	Type of File
ASM	Assembly language source file.
BAK	Backup file.
BAS	File contains a BASIC program in tokenized format.
COM	DOS 2.5 utility program
DAT	Data file.
LST	File containing a program in ASCII format.
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 or the BASIC language interpreter.

Filename Match Characters

In a situation where the same DOS operation is to be performed with several files, a **filename match character** or **wildcard** may be used. For example, it may be necessary to delete all the data files on one diskette, while leaving the program files. Wildcards allow the user to specify a number of files with a single filename. The two wildcards are the asterisk (*) and the question mark (?).

The question mark can stand for any single character, while the asterisk can represent any group of characters. The following example illustrates the use of wildcards. Suppose that the five files listed below are stored on a diskette:

```
TEXT1.DAT
TEXT2.DAT
TEXTY.DAT
TEXT1.BAS
TEXT12.DAT
```

The following filename will match the first three filenames. Here, the question mark matches any single character. Notice that TEXT12.DAT does not match.

TEXT?.DAT

The following filename will match the first and fourth filenames. Here, the asterisk is used to match any file extension.

TEXT1.*

The following filename selects only the BASIC program files. The asterisk is used to match any primary filename with the extension .BAS.

*.BAS

Finally, the universal match uses two asterisks — one for the primary filename and one for the extension. The universal match selects every entry on the disk (a total of five files in our example).

.

Types of Commands

The DOS commands used in the 2.0S ATARI DOS as well as the new Atari 2.5 DOS are all internal commands. This means that the DOS disk need not be present in the drive because the commands have been imbedded in the Atari operating system.

Activating DOS

Generally, DOS must be loaded whenever the computer is powered-up. When starting DOS, the File Management System (FMS) will be read from the DOS diskette into the computer's memory. Then, depending on the specific application, the internal commands of DOS may be loaded automatically.

If BASIC is also to be used, only the FMS will be loaded at first. To load the internal commands, the user must enter an appropriate command. This is the case for most cartridge-based languages. The following steps are involved in loading DOS with BASIC or a cartridge-based language.

1. Power-up the disk drive.
2. Wait for the "busy" light to be extinguished, then insert a DOS diskette.
3. Insert the desired cartridge into the cartridge slot (insert nothing for BASIC).
4. Power-up the system unit (computer).
5. Wait for the language's prompt, then type DOS followed by the RETURN key.
6. The DOS menu will be displayed shortly.

If neither BASIC nor a cartridge is to be used, DOS's internal commands may also be loaded automatically at power-up. If the OPTION key is depressed when the system unit is activated, BASIC will not be utilized and DOS will be loaded and activated. The following steps are involved in loading DOS without BASIC or a cartridge-based language.

1. Power-up the disk drive.
2. Wait for the "busy" light to be extinguished, then insert a DOS diskette.
3. Hold down the OPTION key, while powering-up the system unit.
4. The DOS menu will be displayed shortly.

```
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
SELECT ITEM OR [RETURN] FOR MENU
■
```

```
        DISK OPERATION SYSTEM II VERSION 2.5
              COPYRIGHT 1984 ATARI CORP.
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         P    FORMAT SINGLE
```

DOS Operation

As mentioned previously, a copy of the master diskette should be used in day to day operation. The master diskette should be stored in a safe place. Then, if the back-up DOS diskette becomes damaged or misplaced, additional copies can be made from the master.

Before a copy of the master diskette can be made, the back-up diskette must be formatted. If DOS 2.0S is being used, then menu option (I) will format the disk in the single density format. If an Atari 1050 disk drive and the Atari DOS 2.5 are being used, then it is possible to format in single density (menu option P) or in double density (menu option I).

After the diskette has been formatted a backup of the master diskette can be made using menu option (J). It is also possible to make an operating copy of either DOS by using menu option (H). All of the internal commands will be supported by the copy using either option, however, using option (H) writes only the DOS files and omits any other system boot files (i.e. the 130XE RAM disk on DOS 2.5 or the AUTO-RUN.SYS on DOS 2.0S).

DOS 2.0S

In the following sections, we will discuss DOS 2.0S keyboard usage as well as the various DOS 2.0S commands.

KEYBOARD USAGE

DOS 2.0S uses the ROM-resident operating system's screen editor for command entry. Therefore, the screen can get rather messy after a few commands are used. Pressing a solitary RETURN in response to the DOS prompt will clear the screen then redraw the command menu.

Because the screen editor is used, keyboard usage in DOS 2.0S is essentially identical to keyboard usage in Atari BASIC. Generally, however, the screen editing capability of the editor is not needed, in that most commands can be signaled with a single keystroke. BACK SPACE is usually the only editing key needed.

Once RETURN is pressed following a command entry, the command may be ignored by pressing the BREAK key. At this point, the DOS prompt will be redisplayed.

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 figure 7.1

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 DISK DIRECTORY operation. This file specification consists of the capital letter D followed by the number of the disk drive whose diskettes 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 colon. 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.

```
*DOS        SYS    039
*DUP        SYS    042
 DISP       OBJ    001
 PROGRAM2   BAS    012
 PROGRAM3   BAS    013
 600 FREE SECTORS
```

Figure 7.1. Directory listing

Filename match characters can be used in the filename portion of the SEARCH SPEC parameter. For instance, the following entry would cause all files on drive #2 with the filename extensions 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.

D2:*.DAT

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 send the directory listing to the printer, enter P: as the LIST FILE parameter. For example, the following entry will cause all files on drive #2 with the extension DAT to be listed by the printer.

D2:*.DAT,P:

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 menu, DOS will return control of the Atari computer to the cartridge inserted in the unit. If no cartridge is inserted and BASIC has not been deactivated at power-up, the BASIC prompt will be displayed on the screen.

READY

If a cartridge is not inserted and BASIC has been deactivated, the following message will appear on the screen:

NO CARTRIDGE

Another operation must then be chosen from the menu. This operation or the RESET key may be used to return to BASIC when the MEM.SAV file exists on the diskette. Either procedure will cause data to be correctly returned into memory from the MEM.SAV file. 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 a 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 H. WRITE DOS FILES.

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.

The first time since DOS activation that this operation is selected, a second prompt will appear before the COPY FILE operation is executed. If the user's response to the following 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 FILE operation. The MEM.SAV file will then be retained; however, the copying process will be slower.

TYPE "Y" IF OK TO USE PROGRAM AREA
CAUTION: A "Y" INVALIDATES MEM.SAV

The COPY FILE operation can be used to tack one file to the end of another file. This process is known as **appending**. Suppose that the following two files exist on drive #1.

$\boxed{\text{JOHN}} \longleftarrow$ TEXT1.DAT
$\boxed{\text{DOE}} \longleftarrow$ TEXT2.DAT

After using the append option (/A) of the COPY FILE command, the following two files would exist on drive #1. Notice that the first file listed in the command line is appended to the second file. The append option should not be used with BASIC program files stored with the SAVE command.

```
COPY -- FROM,TO?
TEXT2.DAT,TEXT1.DAT/A
| JOHN DOE | <—TEXT1.DAT
| DOE      | <—TEXT2.DAT
```

D. DELETE FILE

The DELETE FILE operation allows the user to remove unneeded files from the diskette and the disk directory. When 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.

In 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 without query.

A file which has been locked cannot be erased using the DELETE FILE operation. An attempt to do so will generate an ERROR-167 (File locked).

The following example uses the no verification option to erase all files on drive #1. If any of the files on this drive are locked an error will be generated.

Example

```
SELECT ITEM OR RETURN FOR MENU
D
DELETE FILESPEC
*.*/N

SELECT ITEM OR RETURN FOR MENU
```

E. RENAME FILE

The RENAME FILE operation can be used to change the name of any file on the diskette. Be careful not be use RENAME FILE to change the name of DOS.SYS. If DOS.SYS is renamed, the DOS menu will no longer load properly.

When RENAME FILE is specified, the following prompt will appear:

```
RENAME -- GIVE OLD NAME, NEW
```

OLD NAME consists 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 consists 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). Also, 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). If the user attempts to rename a file that does not exist on the diskette, ERROR-170 (File not found) will occur.

Example

```
SELECT ITEM OR RETURN FOR MENU
E
RENAME -- GIVE OLD NAME, NEW
TEXTA,DAT.TEXTB.DAT

SELECT ITEM OR RETURN FOR MENU
```

In the preceding example, TEXTA.DAT on drive #1 is renamed to TEXTB.DAT. In the following example, all files on drive #2 with the extension .BAS will be renamed with the extension .BAK, while retaining their original primary filenames.

Example

```
SELECT ITEM OR RETURN FOR MENU
E
RENAME -- GIVE OLD NAME, NEW
D2:*.BAS,*.BAK

SELECT ITEM OR RETURN FOR MENU
```

F. LOCK FILE

The LOCK FILE operation write-protects a file. Although a locked file can be read, it cannot be written to, renamed, deleted, updated, or appended. In other words, a locked file may not be changed. If an attempt is made to alter a locked file, an ERROR-167 (File locked) will be generated.

When the LOCK FILE operation is specified, the following prompt will appear:

WHAT FILE TO LOCK?

The file specification of the file to be locked should be entered in response to this prompt. Wildcards may be used to lock multiple files with a single file specification. Locked files will appear in the directory listing with an asterisk before the filename. Incidentally, it is good practice to lock the DOS.SYS and DUP.SYS files.

G. UNLOCK FILE

A file can be released from its write-protected state by using the UNLOCK FILE operation. When this operation is specified, the following prompt will appear:

WHAT FILE TO UNLOCK?

The file specification of the file to be unlocked should be entered. Wildcards can be used to specify more than one file.

H. WRITE DOS FILE

The WRITE DOS FILE operation places a copy of DOS onto a diskette. DOS will be copied directly from the computer's memory, not from the diskette during this operation.

First, 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 DOS TO DRIVE 1*

If a Y is entered, DOS will be written to the diskette in the specified drive. Any other entry will abort the WRITE DOS FILE operation.

I. FORMAT DISK

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. Both the 810 and 1050 require approximately two minutes to format a diskette.

When the FORMAT DISK 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 DISK operation.

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. After the third unsuccessful attempt, ERROR-173 (Bad sector 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.

---

* assuming drive #1 was specified in the first prompt.

J. DUPLICATE DISK

The DUPLICATE DISK operation allows an entire diskette to be copied. For example, a back-up DOS diskette could easily be made from the master. This operation can be used with one or more disk drives. When 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 (source), and the drive number containing the diskette on which to place the copy (destination). These should be separated with a comma.

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 disk drive and press RETURN. A portion of the data stored on the diskette will be read into the Atari's memory. Then the following prompt will 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. This process should be continued until the entire diskette has been copied.

If your Atari system contains multiple drives, the duplication process is 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.

It is a good practice to cover the write-protect notch of the source diskette to prevent it from being accidentally overwritten if an error is made.

The following prompt will be displayed the first time a copying procedure is attempted since DOS activation:

```
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, the following prompt will be displayed. FILE is the name of the file to be saved. A drive specifier may be included.

```
SAVE -- GIVE FILE, START, END (,INIT, RUN)
```

The START and END parameters are required for either a binary file or a program. These specify the starting and ending addresses in hexidecimal 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.

Example

```
SELECT ITEM OR RETURN FOR MENU
K
SAVE-GIVE FILE, START, END (,INIT, RUN)
FILEA.OBJ, 2B00, 4C0F

SELECT ITEM OR RETURN FOR MENU
```

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 executed 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 a RUN address must be executed with the RUN AT ADDRESS operation.

Example

```
SELECT ITEM OR RETURN FOR MENU
L
LOAD FROM WHAT FILE?
FILEA.OBJ

SELECT ITEM OR RETURN FOR MENU
```

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.

Example

```
SELECT ITEM OR [RETURN] FOR MENU
M
RUN FROM WHAT ADDRESS
2B00
```

N. CREATE MEM.SAV

The CREATE MEM.SAV operation is used to create a MEM.SAV file on the diskette in drive #1.

When the DOS 2.0S menu is activated, the DUP.SYS file is loaded into memory that is used for BASIC program storage. Whenever a MEM.SAV file is present on the diskette in drive #1, the computer will first transfer all data present in this memory area into the MEM.SAV file. Only then will the DUP.SYS file be loaded. Finally, the DOS menu will appear.

The RUN CARTRIDGE operation or the RESET key may be used to exit DOS. At this time, the program in MEM.SAV will be automatically loaded from MEM.SAV into RAM.

Example

```
SELECT ITEM OR RETURN FOR MENU
N
TYPE "Y" TO CREATE MEM.SAV
Y

SELECT ITEM OR RETURN FOR MENU
```

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

O. DUPLICATE FILE

The DUPLICATE FILE operation is used to copy files from one diskette to another in systems with only one drive. When 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. Wildcards may be used in the filename entry.

The following prompt will be displayed the first time a copying operation (C, J, O) is attempted since DOS activation:

TYPE "Y" IF OK TO USE PROGRAM AREA
CAUTION: A "Y" INVALIDATES MEM.SAV

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.

DUPLICATE FILE will then prompt the user to insert the disk containing the file to be copied (source). The user will next be prompted to insert the destination disk. These two prompts will then alternate until the copy is complete.

INSERT SOURCE DISK, TYPE RETURN
INSERT DESTINATION DISK, TYPE RETURN

DOS 2.5

In the following sections, the revisions and additions to DOS 2.0S, which are present in DOS 2.5 will be discussed.

The DOS 2.5 is a system designed especially for the 130XE computer and/or an Atari 1050 disk drive. If it is used on another computer or disk drive type, the advantage of the DOS will be lessened.

Keyboard Usage

Since the DOS 2.5 is an upgraded version of the DOS 2.0S system all of the main menu commands are the same except for the following changes.

I. FORMAT DISK

In the Atari version 2.5 of DOS, this command formats a disk in the double density format. This will only work with an Atari 1050 disk drive.

P. FORMAT SINGLE

This option, present only on DOS version 2.5, allows the user to format a diskette according to the single density format.

Utility Programs

In addition to the changes in the main menu, several separate programs have been placed on the master diskette which can aid the user in many disk operations. The following is a brief description of these programs.

SETUP.COM

This program is loaded using the (L) Binary load command from the main menu of the DOS. It will run automatically and provide the user with several options. They are menu driven and self explanatory. Their purpose is to change system configurations.

COPY32.COM

This utility allows files created and stored under DOS 3.0 to be copied into the DOS 2.5 (or DOS 2.0) format. The program is loaded using the (L) Binary load option. It is menu driven and self explanatory.

DISKFIX.COM

This is a very useful utility which allows the user to rename files, unerase files, or verify an entire disk. However, care should be used in verifying diskettes as the program will delete any files that are left open. This program is also loaded using the (L) Binary load option, and it is menu driven as well.

RAMDISK.COM

This is the most powerful utility program contained on the DOS 2.5 disk. When the DOS 2.5 is booted, this program is set to automatically run. The program first checks to see if the computer is a 130XE. If the computer is a 130XE it programs the extra 64K of memory into a RAM disk.

The RAM disk represents a remarkable new tool to the user. With it he/she may access the other 64K of memory in the 130XE and use it in the same manner as a disk drive.

In use, the RAM disk runs the same as a disk drive with one major advantage: it is extremely fast. It has 499 sectors, a normal directory and is accessed (i.e. filing procedures and DOS commands) in the same way as a disk drive. The only abnormal characteristic of the RAM disk is its device code, D8.

To use the RAM disk, you need only substitute D8: in any place where a device code is required. In some instances, if a device code is not specified, a default code is used. Such is the case with the DOS 2.5. To use the RAM disk in place of disk drive (D1), the device code of the RAM disk (D8) must be added to the file specifications to override the default drive. For example, to read the directory of the RAM disk:

1. Type DOS then press <RETURN> to enter the DOS 2.5 menu.
2. Type "A" then press <RETURN> to select the disk directory option.
3. When prompted with:

   Directory — Search Spec, List File?

   type in "D8:" and press <RETURN>.

As stated previously, RAM disk files are accessed in the same manner as they are in a disk drive. For example, to load a BASIC program saved in the RAM disk while in BASIC type:

LOAD "D8: Filespec"

and press <RETURN>. The file will load as normal, except much faster.

The only "catch" in using the RAM disk is that the memory it uses to store files is not permanent. In other words, if you turn the computer off without saving your files to permanent memory (disk drive, or tape drive), then they will be lost! Files can be saved by loading them from the RAM disk and saving to the disk drive (or tape drive) or by using the COPY FILE option in the DOS 2.5, menu. For example, to copy a file from the RAM disk to a floppy disk:

1. Type DOS then press <RETURN> to enter the DOS 2.5 menu.
2. Type "C" then press <RETURN> to select the COPY FILE option.
3. When prompted with:

   COPY — FROM, TO?

   type D8: Filespec, D1:Filespec

All other DOS options operate in the same manner on the RAM disk as on the disk drive, however, the correct device code (D8) must be implemented with all of them.

NOTE: If while working with the RAM drive, you discover you need extra memory, it is possible to delete the MEM.SAV, and even the DUP.SYS file. However, upon entering BASIC if you have deleted the MEM.SAV file and then return to the DOS menu, any program in BASIC memory will be lost. Also if the DUP.SYS file is deleted and you return to BASIC it will appear that you can no longer access the DOS menu. What actually is happening is that when DOS <RETURN> is typed the computer searches the RAM disk for DUP.SYS. Because the file has been deleted the computer defaults back to BASIC. To correct this type POKE 5439,ASC("1") <RETURN>. This will direct the computer to search drive one (D1) for the DUP.SYS file in order to run the DOS menu (a disk containing DUP.SYS must be in drive one). To return the control of the DOS menu to the RAM disk, simply replace the DUP.SYS file on the RAM disk and (while in BASIC) type POKE 5439, ASC("8").

As was stated previously, the major advantage of the RAM disk is its speed. It accesses files at speeds that almost make the disk drive obsolete. Almost, that is, but not quite. It is suggested by this author, who has had more than one incidence of a "locked up" computer, that all important files be saved to permanent memory. It may seem an unnecessary burden to do this, until one experiences the pain associated with lost files or programs.

8

Atari BASIC Reference Guide

Introduction

This chapter provides descriptions of the various commands, operations, and functions available in Atari BASIC. The reserved words are listed in alphabetical order with an appropriate abbreviations, if applicable.

The following rules and abbreviations will facilitate our descriptions of the various BASIC commands, operators, and functions.

1. Any capitalized words are keywords.
2. Any words, phrases, or letters shown in lowercase italics identify an entry that must be made by the operator (unless enclosed within brackets).
3. Any items enclosed in brackets [ ] are optional.
4. An ellipsis (...) shows that an item may be repeated as often as desired.
5. Any punctuation marks, except the square brackets (ex. ;,=) must be included where they are shown.
6. If an item is listed directly above another item, either item may be used for correct syntax.

ABS — Function

The ABS function returns the absolute value of its *argument*. A number's absolute value is its value without regard to sign.

Configuration

ABS(*argument*)

argument can be any numeric expression or numeric constant.

Example

```
PRINT ABS(-81), ABS(82)
81                82
```

ADR

Function

The ADR function returns the absolute memory address of the *argument*. The *argument* must be a predimensioned 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 may be used to locate the string.

Configuration

ADR(*argument*)

Example

```
X = USR(ADR("Lw d "))
```

The previous command line will **reboot** or cold start the Atari. Typing this line is equivalent to flipping the power switch off, then on. Upon execution, this command will erase any RAM-resident program and will cause the Atari to behave as if it had just been powered up.

The string argument of the command line is the machine language command to cold start the Atari. The USR function executes this command by finding its address using ADR.

AND

Operator

AND is a logical operator. This reserved word is generally used to combine two comparisons in the context of an IF...THEN statement.

Configuration

expression1 AND *expression2*

If an *expression* is non-zero, that *expression* will be evaluated as true. Likewise, an *expression* with a value of zero will be evaluated as false. The following is the truth table for AND.

X	Y	X AND Y
true	true	true
true	false	false
false	true	false
false	false	false

In Atari BASIC, a true is represented by a 1 and a false by a 0.

Example 1

```
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, *expression1* is true and *expression2* is false. This corresponds to the second line of the truth table. The result from the table is false (0), so the condition of the IF...THEN statement is false, and the next line is executed.

Example 2

```
PRINT (3 = 1 + 2) AND (-5)
1
```

In this example, 3 is compared to the result of 1 + 2, so the first *expression* evaluates as true. The second *expression* (–5) is non-zero, so it is also evaluates as true. According to the AND truth table, if both expressions evaluate as true, then the whole expression is true. Therefore, 1 is printed.

ASC

Function

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

ASC(*argument*)

Example

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

ATN

Function

The ATN function is a trigonometric function that returns the arc-tangent of its *argument*. The *argument* can be a numeric expression or numeric constant in radians. The value returned will be the primary angle in radians, unless degrees have been specified with DEG ($-\pi/2 < \text{angle} < \pi/2$; $-90° < \text{angle} < 90°$).

Configuration

ATN(*argument*)

Example

```
10 PI = 4 * ATN(1)
20 PRINT PI
RUN
3.14159267
```

In the preceding example, the arctangent of 1 returns the value $\pi/4$. Multiplying this value by 4 returns the indicated value.

BYE — Statement

BYE switches the system to the Self-Test mode. The system will then perform the various user specified self-tests. System control will be returned to BASIC when the RESET key is pressed. BYE will erase any RAM resident program.

Configuration

BYE

Example

```
10 BYE
RUN
```

CHR$ — Function

The CHR$ function returns the character with the ASCII code specified by *argument*. Although *argument* values can range from 0 to 65535, the ASCII code corresponding to *argument* modulo 256 is used.

Configuration

CHR$(*argument*)

Example

```
10 PRINT CHR$(65)
20 PRINT CHR$(65 + 256)
RUN
A
A
```

CLOAD (CLOA.) — Statement

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

```
10 CLOAD
RUN
```

CLOG — Function

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

Configuration

CLOG(*argument*)

Example

```
PRINT CLOG(4)
0.602059991
```

CLOSE (CL.) Statement

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

The *filenumber* of a CLOSE statement must be identical to the *filenumber* used in the corresponding OPEN statement. A *filenumber* that has been opened for the use of a particular I/O device must be closed before it can be used for another device. *filenumber* can be any numeric constant or expression.

Configuration

CLOSE #*filenumber*

Example

```
CLOSE #3
```

CLR Statement

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 redimensioned. CLR also frees any memory used by dimensioned variables.

Configuration

CLR

Example

```
10 PRINT FRE(0)
20 DIM A(500)
30 PRINT FRE(0)
40 CLR
50 PRINT FRE(0)
RUN
13246
10240
13246
```

COLOR

Statement

The COLOR statement determines the data that will be placed on the screen by subsequent PLOT statements. In the text mode (0), COLOR determines which character will be plotted. In the character graphics modes (1, 2, 12, 13), COLOR determines the character as well as its color. In the bit-image graphics modes (3-11, 14, 15), COLOR determines the color of any subsequently plotted pixels.

Configuration

COLOR *argument*

In all graphics modes, the *argument* of the COLOR statement must be non-negative. If it is not an integer, it will be rounded off.

In mode 0, the text and background are displayed in the same color but in differing brightnesses. The color, the brightness of the text, and the brightness of the background are determined by the SETCOLOR command. The COLOR statement does not select a color; it indicates the character to be printed with the next PLOT statement. Table 8.1 lists these characters and their corresponding COLOR statement *arguments.*

If two characters are assigned the same numeric representation, the character that is displayed on the screen depends on the value stored in memory location 756. The character on the right corresponds to a value of 224 (standard), while the character on the left corresponds to a value of 204 (extended).

POKE 756,224	set standard
POKE 756,204	set extended

Example 1

```
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; each time the COLOR statement has a different value as its *argument*. It can be seen from table 8.1 that in graphics mode 0, COLOR 65 indicates the character A.

After a COLOR 65 statement has been executed, any PLOT or DRAWTO statement will output the character "A" until another COLOR statement has been executed.

Example 2

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

When executed, the preceding program will print the character "A" in the upper left corner of the screen because of the PLOT 0,0 statement. The DRAWTO 10,10 will cause a diagonal line consisting of several A's to appear on the display. A's will appear at the positions (0,0), (1,1), (2,2)...(10,10). Line 50 sets the screen color to orange.

Table 8.1. Characters displayed by COLOR statement values in graphics mode 0

Character ext.	Character std.	COLOR Value Normal/ Inverse
		0/128
		1/129
		2/130
		3/131
		4/132
		5/133
		6/134
		7/135
		8/136
		9/137
		1-/138
		11/139
		12/140
		13/141
		14/142
		15/143
		16/144
		17/145
		18/146
		19/147
		20/148
		21/149
		22/150
		23/151
		24/152
		25/153
		26/154
		27/---
		28/156
		29/157
		30/158
		31/159
		32/160
!	!	33/161
"	"	34/162
#	#	35/163
$	$	36/164
%	%	37/165
&	&	38/166
'	'	39/167
(	(	40/168
)	)	41/169
*	*	42/170
+	+	43/171
,	,	44/172
-	-	45/173
.	.	46/174
/	/	47/175
0	0	48/176
1	1	49/177
2	2	50/178
3	3	51/179
4	4	52/180
5	5	53/181
6	6	54/182
7	7	55/183
8	8	56/184
9	9	57/185
:	:	58/186
;	;	59/187
<	<	60/188
=	=	61/189
>	>	62/190
?	?	63/191
@	@	64/192
A	A	65/193
B	B	66/194
C	C	67/195
D	D	68/196
E	E	69/197
F	F	70/198
G	G	71/199
H	H	72/200
I	I	73/201
J	J	74/202
K	K	75/203
L	L	76/204
M	M	77/205
N	N	78/206
O	O	79/207
P	P	80/208
Q	Q	81/209
R	R	82/210
S	S	83/211
T	T	84/212
U	U	85/213
V	V	86/214
W	W	87/215
X	X	88/216
Y	Y	89/217
Z	Z	90/218
[	[	91/219
\	\	92/220
]	]	93/221
^	^	94/222
_	_	95/223
	♦	96/224
a	a	97/225
b	b	98/226
c	c	99/227
d	d	100/228
e	e	101/229
f	f	102/230
g	g	103/231
h	h	104/232

table 8.1 continued on next page

Table 8.1. (cont.) Characters displayed by COLOR statement values in graphics mode 0.

Character ext.	Character std.	COLOR Value Normal/ Inverse	Character ext.	Character std.	COLOR Value Normal/ Inverse	Character ext.	Character std.	COLOR Value Normal/ Inverse
i	i	105/233	r	r	114/242	[illegible]	♠	123/251
j	j	106/234	s	s	115/243	│	│	124/252
k	k	107/235	t	t	116/244	Clear	Screen	125/---
l	l	108/236	u	u	117/245	▶	▶	126/254
m	m	109/237	v	v	118/246	◀	◀	127/255
n	n	110/238	w	w	119/247		EOL	---/155
o	o	111/239	x	x	120/248	[illegible]	[illegible]	---/253
p	p	112/240	y	y	121/249			
q	q	113/241	z	z	122/250			

The COLOR statement has an additional function in graphics modes 1 and 2. Besides character selection, COLOR must also specify the color of the character. Table 8.2 lists the values of the COLOR statement *arguments* for each character. Each character can be printed in one of four colors. The columns of the table correspond to the color registers 1-4. The standard character set will be used unless either the alternate or the extended character set is specified by an appropriate POKE statement.

POKE 756,224	set standard
POKE 756,226	set alternate
POKE 756,206	set extended

Table 8.2. Standard, alternate and extended character sets in graphics modes 1 and 2 and color register values

Character			Value for Color Register			
Std.	Alt.	Ext.	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	22	171	139
,			44	12	172	140
-			45	13	173	141
.			46	14	174	142
/			47	15	175	143
0			48	16	176	144
1			49	17	177	145
2			50	18	178	146
3			51	19	179	147
4			52	20	180	148
5			53	21	181	149
6			54	22	182	150
7			55	23	183	151
8			56	24	184	152
9			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 8.2 contined on next page

Table 8.2. (cont.) Standard, alternate and extended character sets in graphics modes 1 and 2 and color register values

Character			Value for Color Register			
Std.	Alt.	Ext.	0	1	2	3
@	♦	¡	64	96	192	224
A	a	a	65	97	193	225
B	b	b	66	98	194	226
C	c	c	67	99	195	227
D	d	d	68	100	196	228
E	e	e	69	101	197	229
F	f	f	70	102	198	230
G	g	g	71	103	199	231
H	h	h	72	104	200	232
I	i	i	73	105	201	233
J	j	j	74	106	202	234
K	k	k	75	107	203	235
L	l	l	76	108	204	236
M	m	m	77	109	205	237
N	n	n	78	110	206	238
O	o	o	79	111	207	239
P	p	p	80	112	208	240
Q	q	q	81	113	209	241
R	r	r	82	114	210	242
S	s	s	83	115	211	243
T	t	t	84	116	212	244
U	u	u	85	117	213	245
V	v	v	86	118	214	246
W	w	w	87	119	215	247
X	x	x	88	120	216	248
Y	y	y	89	121	217	249
Z	z	z	90	122	218	250
[	♠	Ä	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.

Example 3

```
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
```

Example 3 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 8.2, 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	LIGHT GREEN
2	DARK BLUE
3	RED
4	BLACK

In example 3, 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 lowercase 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 (see table 8.2). As a result, an orange "heart" will appear in every space except where the word ATARI appears.

In graphics modes 3-7, 10, 14, and 15, the COLOR statement is used to choose the color register that will be used to plot points and draw lines. These modes are different from modes 0 through 2 because mode 0, 1, and 2 are used to place characters on the screen. Modes 3-7, 10, 14, and 15 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. Here, the COLOR statement actually chooses a color register, not a character.

Modes 3, 5, 7, and 15 can display four colors simultaneously. The *argument* of the COLOR statement is used modulo 4. COLOR 0 selects the color stored in color register 4; COLOR 1 selects color register 0; COLOR 2 selects color register 1; and COLOR 3 selects color register 2.

Example 4

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

Example 4 displays the four 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, 7, and 15

COLOR's *argument*	**DEFAULT COLOR**	**COLOR REGISTER**
0	BLACK	4
1	ORANGE	0
2	LIGHT GREEN	1
3	DARK BLUE	2

Since COLOR selects a color register and not an actual color, the SETCOLOR command can be used to change the default colors to user specified hues. See the SETCOLOR command for details.

In graphics modes 4, 6, and 14, the COLOR statement selects a color register as in modes 3, 5, 7, and 15. However, modes 4, 6, and 14 support only two colors. If an *argument* greater than 1 is specified, *argument* modulo 2 will be used. In other words, an even *argument* will select color register 4, while an odd *argument* will select color register 0.

GRAPHICS MODES 4, 6, 14

COLOR's *argument*	**DEFAULT COLOR**	**COLOR REGISTER**
0	BLACK	4
1	ORANGE	0

When configured in mode 10, the Atari can display nine colors simultaneously. The COLOR statement is again used to specify a color register. The screen usually has 5 color registers alloted to it. However, mode 10 makes use of the 4 player-missile color registers as well. These are located at memory locations 704 to 707 and must be accessed by POKE's.

GRAPHICS MODE 10

COLOR's *argument*	COLOR REGISTER (MEMORY LOCATION)
0	- (704)
1	- (705)
2	- (706)
3	- (707)
4	0 (708)
5	1 (709)
6	2 (710)
7	3 (711)
8	4 (712)
9	4 (712)
10	4 (712)
11	4 (712)
12	0 (708)
13	1 (709)
14	2 (710)
15	3 (711)

The locations 704 to 707 are not reset by BASIC or the operating system. Therefore, these color registers have no "default" color assigned to them. Generally, these locations have a value of zero (black) until intentionally changed.

Example 5

```
10 GRAPHICS 10
20 FOR I = 0 TO 67
30 COLOR I/8
40 PLOT I,0
50 DRAWTO I,159
60 NEXT I
70 POKE 704,222:REM YELLOW-GREEN
80 POKE 705,126:REM LIGHT BLUE
90 POKE 706,190:REM GREEN-BLUE
100 POKE 707,200:REM GREEN
110 GOTO 110
```

Graphics mode 8 has only one color, with two luminence levels. As a result, the COLOR statement is used to select the luminence 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. If the *argument* is specified greater than 1, *argument* modulo 2 will be used.

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.

Graphics modes 9 and 11 differ from the other bit-image modes in that the COLOR statement actually specifies a color, not a color register. In mode 9, only one hue may be displayed although all 16 shades (luminences) of that hue may be shown simultaneously. In mode 11, only one luminence may be displayed although all 16 hues in that shade may be shown.

The following table summarizes the hue selection in mode 11 and the luminence selection in mode 9. If *argument* is greater than 15, *argument* modulo 16 will be used.

GRAPHICS MODES 9 and 11

COLOR's *argument*	**MODE 9**	**MODE 11**
0	darkest	gray
1	↑	gold
2			orange
3	darker	red-orange
4	↑	red
5			red
6	dark	purple-blue
7	↑	blue
8	↓	blue
9	bright	light blue
10			turquoise
11	↓	green-blue
12	brighter	green
13			yellow-green
14	↓	orange-green
15	brightest	light orange

Example 6

```
10 GRAPHICS 11
20 SETCOLOR 4,0,10
30 FOR I = 0 TO 79
40 COLOR I/5
50 PLOT I,0:DRAWTO I,159
60 NEXT I
70 GOTO 70
```

In mode 9, color register 4 selects the hue of the display; likewise, in mode 11, color register 4 selects its luminence.

Graphics modes 12 and 13 are somewhat an enigma. These are color character graphics modes; however, their color selection process is not similar to that of modes 1 and 2. COLOR, in modes 12 and 13 specifies a character which in turn determines the color of the displayed data. In other words, the character determines its own color; the COLOR statement does not.

The Atari does not have a built-in character set which is compatible with these modes. Unless the user is interested in defining his own character set, modes 12 and 13 are generally not useful.

COM — Statement

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

Configuration

COM *variable*(*range*[,*range*]) [,*variable*(*range*[,*range*])]...

Example

```
10 COM PAT$(81), KAREN(84)
```

CONT (CON.)

Statement

The CONT statement causes a program which had been stopped to continue execution at the next numbered line. A program will be stopped because of an error, 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, or 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 RESET, but this will generally have negative results for the following reason: All I/O will be closed; the screen will be cleared; graphics mode 0 will have resumed; etc.

Configuration

CONT

Example

```
10 PRINT "PUPPY"
20 STOP
30 PRINT "LOVE"
RUN
PUPPY

STOPPED AT LINE 20
CONT
LOVE

READY
```

In the preceding example, the computer's responses are set in bold type to differentiate them from user entered lines.

COS — Function

The COS function returns the cosine of its *argument*. The *argument* will be assumed in radians unless a DEG statement precedes the COS statement. In this case, the *argument* is assumed in degrees.

Configuration

COS(*argument*)

Example

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

CSAVE (CS.) — Statement

The CSAVE command is used to copy the program in the computer's memory onto 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 filenumber 7 had been open for another device, an error will occur, but the file will be closed. A repeat of CSAVE will then be successful.

Configuration

CSAVE

CSAVE may be used as a program line, although this is rarely done.

Example

```
10 CSAVE
```

DATA (D.)

Statement

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.

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

DATA *constant* [,*constant*]...

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

Example 1

```
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 Error-8 (Input statement error). Numeric data must not include commas.

Example 2

```
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.

Ordinarily, data can only be read once. A RESTORE statement may be used to alter the data reading sequence or to reread data if necessary.

DEG (DE.) Statement

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 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 had not been present, the result would have been 0.893997024.

DIM (DI.) Statement

The DIM statement is used to set aside memory space for strings, arrays or matrices.

Configuration

DIM *variable* (*range*[,*range*])[,*variable*(*range*[,*range*])]...

A DIM statement can include any combination of numeric and string variable dimension statements. For example, the following statement dimensions four *variables* in one statement:

```
DIM A(10,10),B(69),A$(255),B$(10)
```

A string *variable* can contain only a single string. The *range* of a string variable indicates the maximum number of characters that the string 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
```

ERROR-9 is the string dimension error. This error also occurs if a *variable* is dimensioned twice in the same program (without an intervening CLR).

The maximum size of a string *variable* depends on the amount of available memory at the time of the DIM statement. Also, a string's length may not exceed 32767 characters.

Dimensioning a numeric *variable* determines the number of elements that the *variable* can contain. Each element is an independent entity that may take a value from -9.9×10^{97} to $+9.9 \times 10^{97}$. The following example shows how to assign four values to a subscripted *variable*:

Example

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

Notice that four values can be assigned to a *variable* that has a *range* 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 *range* 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:

X

	0	1	2
0			
1			
2			
3			

DOS (DO.) Statement

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 self-test mode. To return to BASIC from the self-tests, press RESET.

Configuration

DOS

When the DOS command is executed, all I/O is closed except filenumber 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 the disk functions. There are three versions of the Disk Operating System, version 1.0, version 2.0S, and version 2.5. The DOS command has a different effect in each of the three 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. FUN 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 2.0S, 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 RESET key.

```
DISK OPERATING SYSTEMS II VERSION 2.5
COPYRIGHT 1984 ATARI CORP.
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. DUPLCIATE FILE
H. WRITE DOS FILES     P. FORMAT SINGLE
```

In DOS 2.5, as in DOS 2.0, the DOS consists of two files. However, if DOS 2.5 is used on the 130XE with the RAM disk program, a disk with DUP.SYS need not be in drive one. The computer will use the DUP.SYS file on the RAM disk.

In addition, the MEM.SAV need not be present in drive one if it is present on the RAM disk. All other operations are explained in chapter 7 of this text.

DRAWTO (DR.) — Statement

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 *column,row*

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 24 rows, numbered 0 through 39 and 0 through 23, respectively. DRAWTO 40,20 would result in ERROR-141. DRAWTO 40,20 contains an argument that lies outside the range of the 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 1

```
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 preceding example began by plotting a point at line 30, then proceeded to draw the four sides of a square in lines 40, 50, 60, and 70.

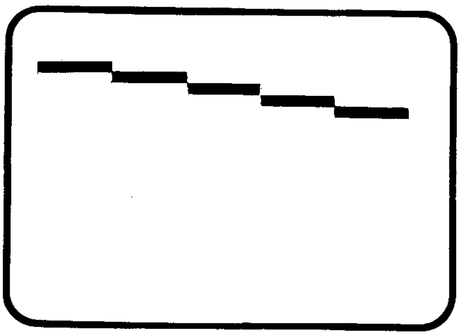
The DRAWTO statement can also be used in graphics modes 0, 1, and 2. However, the PLOT statement in the text modes (0, 1, and 2) places a character on the display. The COLOR statement determines the character that is printed. As a result, the DRAWTO statement in the text mode creates a line of characters.

Example 2

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

Example 2 specifies graphics mode 2 in line 10. Line 20 indicates the character that appears on the display. 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

Statement

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 conclude BASIC programs with an END statement.

Configuration

END

When an END statement is executed, all I/O will be closed except filenumber 0, and all sound will be turned off. Also, if a full screen graphics mode had been active, graphics mode 0 will be activated.

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

Statement

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 "*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 contains line numbers 10, 20, 30..., and the program being loaded (using ENTER) contains line numbers 5, 15, 25, 35,..., the resulting program in RAM will 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 a drive other than drive #1 is being used, the number of the drive must be specified.

Example

```
ENTER "C"
ENTER "D2:JONES"
```

EXP — Function

The EXP function returns the exponential of the *argument*. The exponential is the value of e (approximately 2.71828179) raised to the power of the *argument*.

Configuration

EXP(*argument*)

Example

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

FOR (F.)...NEXT (N.) Statement

The FOR...NEXT statements are used to execute a sequence of commands a set number of times.

Configuration

FOR *variable* = *start* TO *stop* [STEP *increment*]
.
.
.
NEXT *variable*

variable is a numeric variable that is used as a counter. *start, stop,* and *increment* are numeric expressions or constants. *start* is the initial value of the counter and *stop* is the final value. The counter is increased or decreased depending on the sign of *increment*. If *increment* is omitted, it will be assumed as 1. Every FOR statement must have a corresponding NEXT statement.

The program lines following the FOR statement will be executed until the NEXT statement is encountered. At this point, the counter's value is increased by the STEP value. The value of the counter is then compared with its final value. As long as the counter's value does not exceed *stop* (assuming a positive *increment*), the program will branch back to the statement following the FOR statement. The entire process will then be repeated.

Example 1

```
100 FOR I = 1 TO 5
200 PRINT I;
300 NEXT I
RUN
12345
```

In the previous example, the FOR...NEXT loop is repeated five times. Although line 200 is the only statement inside the loop, any number of program lines could have been placed there.

In line 100, I is assigned the value 1. When the NEXT I statement is executed, the program returns to the FOR statement with its value incremented by one. This loop is repeated until I is set equal to 6. When the counter is set equal to 6, note that the body of the loop is not executed. The program will proceed with the statement following NEXT I. In the preceding example, no program lines follow the NEXT statement; therefore, program execution halts.

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

Example 2

```
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 3

```
10 FOR K = 10 TO 5 STEP -2
20 PRINT K,
30 NEXT K
40 PRINT
50 PRINT K
RUN
10       8       6
4
```

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 returns to line 10. The current value of the counter is decremented by 2. The new value of K is 4. This K value is less than the stop value of 5, so execution of the program branches to the statement immediately following the NEXT statement in line 30.

If the counter of a loop is being incremented, the loop will be executed until the counter exceeds the final value. For example, FOR J=1 TO 4 STEP 2 would cause the body of the loop to be executed twice. The final value of J would be 5.

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 4

```
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. The statement POP facilitates exiting a FOR...NEXT loop prematurely.

FRE — Function

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

Configuration

FRE(*argument*)

Example

```
PRINT FRE(0)
10109
```

GET (GE.) — Statement

The GET function reads 1 byte from a channel that has been opened for input. GET is used with the keyboard, display, cassette unit, disk drive, RS232 port, and printer.

Configuration

GET #*filenumber, variable*

filenumber indicates the data channel that will be used. This channel must be previously specified in an OPEN statement. If *filenumber* is not an integer, it will be rounded off. *variable* will be assigned the value read from the channel. This value will be an integer between 0 and 255.

Example 1

```
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 data channel and specifies *filenumber* 3 for input with the cassette unit. *filenumber* 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 data file.

GET is used with the disk in the same fashion as it is used with the cassette unit. However, the OPEN statement must include a file specification. The first argument of the OPEN statement is a *filenumber.* 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), 12 (input and output), or 13 (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 the channel that is not already OPEN. 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 -- ex. CONTROL-A) is pressed, the ASCII code of the character is assigned to the variable in the GET statement.

Example 2

```
10 OPEN #3, 4, 0, "KEYBOARD"
20 GET #3, CHAR
30 PRINT CHR$(CHAR);
40 GOTO 20
```

The preceding example consists of a program that uses the GET statement with the keyboard. Line 10 opens *filenumber* 3 for keyboard input. In line 20, the GET statement assigns the ASCII value of a character to the variable CHAR. Line 30 displays the character on the screen. When the program is executed, line 10 opens the I/O channel, then the program waits at line 20. When a keystroke occurs, the program continues.

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 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. The cursor advances to the next position after a GET statement has been executed.

Example 3

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

Example 3 consists of a program that uses GET with the display. Line 10 opens *filenumber* 3 for input from the display (device "S"). Line 20 specifies graphics mode 2. Line 30 indicates the character and color that is displayed. COLOR 65 indicates an uppercase 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 *filenumber* in the GET statement must be the same as the *filenumber* in the OPEN statement. Line 70 displays the COLOR information (65) on the display, and line 80 closes the I/O channel.

Whenever a GRAPHICS command is executed, Atari BASIC opens *filenumber* 6 to the screen device. Since the screen is already opened on *filenumber* 6, it need not markedly be reopened under a new *filenumber*. Example 3 could be revised as follows:

Example 4

```
20 GRAPHICS 2
30 COLOR 65
40 PLOT 0,0
50 POSITION 0,0
60 GET #6,X
70 PRINT X
80 END
```

GET can also be used with the screen editor (device "E"). The OPEN statement must include an unused I/O *filenumber*. 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 5

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

In the preceding example, line 10 opens *filenumber* 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.

If the editor is opened in the forced-read mode, the GET statement does not require that RETURN be pressed. The forced-read mode is activated when the operation code of the OPEN statement is 5 (input) or 13 (input and output).

Example 6

```
10 OPEN #3, 5, 0, "EDITOR"
20 PRINT "A"
30 POS 2,0
40 GET #3, X
50 PRINT X
60 CLOSE #3
RUN

65
```

GOSUB (GOS.) Statement

GOSUB branches program control to the subroutine beginning at the *linenumber* specified by its argument.

Configuration

GOSUB *linenumber*

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 preceding example illustrates a subroutine that is called four times, from two 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 three 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.

GOTO (G.) — Statement

The GOTO statement causes the program to proceed at the indicated *linenumber.*

Configuration

GOTO *linenumber*

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.

GRAPHICS (GR.) — Statement

GRAPHICS sets one of the graphics modes.

Configuration

GRAPHICS *argument*

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 1-8, 12-15 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...THEN

Statement

The IF...THEN statement exploits the decision making power of your computer by setting up a condition that will influence the program flow.

Configuration

IF *expression* THEN *statement* [:*statement*]...

The *expression* that follows IF can be either logical or algebraic. Any non-zero algebraic expression is considered true. *statement* can be any valid BASIC statement. If *expression* is evaluated as true, then *statement* will be executed. If *statement* is a number then a GOTO that line number will be executed (assumed GOTO).

Example 1

```
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 preceding 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 2

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

Example 2 shows that statements can follow a THEN statement, separated by colons. If the *expression* is true, the *statements* are executed. If the *expression* 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 *expression* 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 3

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

The preceding example contains a program that ends when the *expression* is true. The *expression* 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 *expression* 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 *expression* 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 *expression* 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.) Statement

The INPUT statement permits data entry while the program is being executed.

Configuration

INPUT *variable* [,*variable*]...

When an INPUT statement is executed, program execution will stop temporarily. A question mark will be displayed on the screen. The user may then enter the desired data at the keyboard. This data is assigned to the *variable(s)* listed in the INPUT statement.

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.

Example 1

```
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).

Example 2

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

In the preceding 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$. When the program is executed, the INPUT statement causes the program to wait at line 20 for input. The user responds with two data items. The value 45 is entered for a value of X. The string value JONES, BILL is entered for a value of X$. These two data items could be entered on separate lines. Notice that the comma in the string value does not separate data items.

INPUT# (I.#) Statement

The INPUT# statement is used to read data items from a sequential file or device and to assign those items to a program *variable.*

Configuration

INPUT# *filenumber, variable* [,*variable*]...

filenumber is the number assigned to the file specification when it was OPEN'ed. *variable* is the name of the variable that will be assigned a data item from the device or file. The data items being read and assigned to the *variable(s)* may either be from a sequential file on diskette or cassette; from the keyboard; from the screen; or from the interface module.

The INPUT# statement can be used with the cassette unit to recover data. When the cassette unit is used, an OPEN statement must be executed before an INPUT# statement is encountered. The OPEN statement must include a *filenumber,* the operation code for input (4), and the device code ("C"). The third argument of the OPEN statement is a special function code, and is generally set to zero. If any of the arguments of an OPEN statement are not integers, they will be rounded off.

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

Example 1

```
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 filenumber 1 for input from the cassette unit. When line 20 is executed, the tone sounds to remind the operator to find the correct position on the tape, press PLAY on the cassette drive, 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 data file.

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 a *filenumber* and *variable* names.

The OPEN statement for the data channel must include the *filenumber* and the operation code 4 (input), 6 (directory), 12 (update), or 13 (special update). The third argument of the OPEN statement is zero, and the fourth argument is the file specification.

Example 2

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

If only one drive is in use, the device name is simply "D". If two or more drives are being used, the number of the drive must be specified.

The INPUT# statement can also be used with the keyboard. The OPEN statement must include a *file number,* operation code 4, auxiliary byte 0, and the device "K".

Example 3

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

Example 3 contains a program that uses the keyboard for input. Line 10 dimensions the variable Y$. Line 20 opens *filenumber* 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. Line 40 displays the values of the two variables, and line 50 closes the *filenumber.*

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.

INPUT# operates in a similar manner with the screen editor (E:), screen device (S:), and RS232 module (R:). INPUT# will continually read data bytes from a file or device until a carriage return is encountered. These bytes are stored in the specified *variable.*

INT

Function

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

Configuration

X = INT(*argument*)

Examples

```
PRINT INT(13.9)
13
PRINT INT(-4.7)
-5
```

LEN

Function

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
11
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.) — Function

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

Configuration

[LET] *variable* = *expression*

Example

```
10 A = 4
20 LET COLOR = 5
30 PRINT COLOR, A
RUN
5          4
```

Notice that the LET in line 20 is **not** optional, as is usually the case. The BASIC interpreter would not accept the line without the LET, because COLOR is a reserved word in Atari BASIC. If the interpreter saw the following command, it would assume that a COLOR command of incorrect syntax was entered:

```
20 COLOR = 5
20 ERROR-   COLOR ■5
```

LIST (L.)

Statement

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

Configuration

LIST [*"filespec",*][*linenumber*[*,linenumber*]]

The LIST statement can be used to save a program, or part of a program, on a disk or cassette in the file indicated by *filespec.* The ENTER statement is the only Atari BASIC statement that can recover a program saved with LIST. The optional *linenumber(s)* indicate the section of the program that is to be saved. If no *linenumber* is specified, the entire program will be saved. If only one *linenumber* is specified, only that line of the program will be saved. If both *linenumbers* are specified, the section of the program between those lines is saved, inclusively. That is, if either or both of the specified *linenumbers* are contained in the program, they will also be 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 cassette drive, 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 "D *number:filename"* followed by the appropriate *linenumbers* (if any).

Example 1

```
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 *linenumbers* 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 to reference drive #1. To reference a drive other than drive #1, the number of the drive must also be specified (ex. D2:PAT, D3:REBEL).

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 *linenumbers*.

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

Example 2

```
LIST 20

20 FOR A = 1 TO 100

READY
```

If both *linenumbers* are specified, those two lines are displayed along with all the code between those lines. If either or both of the specified *linenumbers* do not appear in the program, the section of the program between those linenumbers 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 printer, or course, must be on-line.

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 3

```
PRINT "↑" (ESC, 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 4

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.) — Statement

The LOAD statement can be used to recover programs recorded on diskette or cassette tape with the SAVE statement.

Configuration

LOAD *"filespec"*

When the LOAD statement is executed, the computer's memory will be cleared before the new program is loaded. Also, all I/O (except filenumber 0) will be closed, and the voices shut off.

With the cassette unit, LOAD does not require a filename; only device name is necessary ("C"). When the LOAD "C" statement is executed, a single tone will sound to remind the operator to align the tape and press PLAY on the cassette unit. Pressing the RETURN key will start the retrieval process.

With a disk drive, the LOAD statement must include a filename along with the device name. When referencing any drive but drive #1, the device name must also include the number of the drive. If drive #1 is being referenced, the device name "D:" is sufficient.

Example

LOAD "D2:GRADES"

LOCATE (LOC.) — Statement

The LOCATE statement is used to obtain COLOR data from the screen. This data will be returned through a numeric *variable.*

Configuration

LOCATE *column,row,variable*

column and *row* are numeric expressions that determine from where on the screen the COLOR data is to be obtained. *variable* will be assigned this data value.

A LOCATE statement is equivalent to the following two commands. For this reason, LOCATE may not be used unless a GRAPHICS command has previously been executed.

POSITION *column, row*
GET #6, *variable*

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 a graphics mode 3. Line 20 indicates which color register 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 data at that position is 2. The LOCATE statement assigns the COLOR data 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.

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 — Function

The LOG function returns the natural logarithm of the *argument*. The natural log function is undefined for *arguments* less than or equal to zero. Therefore, a value error results from a zero or negative *argument*.

Configuration

LOG (*argument*)

Examples

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

LPRINT (LP.) — Statement

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

Configuration

LPRINT [*expression*] [; ,] [*expression*]...

The LPRINT statement can include numeric variables 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 commas or semicolons. 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 an LPRINT statement. If a semicolon is used at the end of an 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 filenumber 7. If filenumber 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 — Statement

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 files except filenumber 0.

Configuration

NEW

NEXT (N.) — Statement

The NEXT statement is always used in conjunction with a FOR statement to form program loops. See the FOR statement for more information.

Configuration

NEXT *variable*

NOT Operator

The NOT operator logically compliments the value given in *expression*. It is generally used in an IF...THEN statement.

Configuration

NOT *expression*

expression is a numeric constant or numeric expression. If the *expression* evaluates to true (non-zero), false (zero) will be returned. If the *expression* evaluates to false (zero), true (one) will be retained.

Example

```
10 X = 2
20 IF NOT(X = 1) THEN PRINT "X" DOES NOT EQUAL ONE"
30 END
RUN
X DOES NOT EQUAL ONE
```

NOTE (NO.) Statement

The NOTE statement returns the location of the file pointer for a specified disk file. The NOTE statement is not available in version 1.0 of the disk operating system although it is supported in versions 2.0S and 2.5.

Configuration

NOTE #*filenumber* , *variable1*, *variable2*

The NOTE statement must specify a *filenumber* that is presently opened to a disk file.

With DOS 2.0S and DOS 2.5, the second argument is a numeric variable that will be assigned the sector number of the file pointer. The third argument is a numeric variable that will be assigned the byte number of the file pointer within the specified sector.

ON...GOSUB,ON...GOTO

Statement

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

ON *expression* GOSUB *linenumber* [, *linenumber*]...
ON *expression* GOTO *linenumber* [, *linenumber*]...

The control *expression* determines to which *line number* the program will proceed. If the control *expression* equals 1, the program branches to the first *linenumber* after the GOTO or GOSUB. If the control *expression* equals 2, the program branches to the second *linenumber* after GOTO or GOSUB, etc.

The control *expression* must evaluate between 0 and 255 to prevent an error. If *expression* evaluates to zero or to a value greater than the number of *linenumbers* specified, the program line following the ON statement will be executed.

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 number. The second choice is line 40, so the message SECOND is printed.

OPEN (O.) — Statement

The OPEN statement is used to open an input/output *filenumber* for an input or output device. The computer cannot receive input from or send output to a device unless an I/O *filenumber* has been opened for that purpose.

Configuration

OPEN #*filenumber,aux1,aux2, "filespec"*

The first argument of an OPEN statement is the *filenumber*. *file-number* can range from 0 through 7. *filenumber* 0 is always reserved for the editor. *filenumber* 6 is used for graphics, while *filenumber* 7 is used to save and load programs. *filenumber* 7 is also used with the LPRINT statement.

As a result, *filenumbers* 1 through 5 are available for use with BASIC programs. *filenumber* 6 and 7 are available only on a limited basis for use with BASIC programs. *filenumber* 6 is available if no graphics are used. *filenumber* 7 is available unless programs are being loaded or saved. Also, *filenumber* 7 is unavailable if an LPRINT statement is executed.

aux1 indicates the operation of the input/output device. In general, *aux1* = 4 if the computer is accepting information (input). Generally, *aux1* = 8 if the computer is sending information (output) to a device. Table 8.3 contains a list of the I/O operations with their associated devices and operation numbers.

Table 8.3 is not complete because the screen device has been discounted. The use of the OPEN statement concerning the screen device will be discussed in the latter part of this section.

Table 8.3 I/O operations

Device	Operation Number (aux1)	Operation Type
Cassette unit	4 8	input output
Keyboard	4	input
Printer	8	output
Editor	4 5 8 12 13	input:keyboard input:screen (forced read) output:screen input:keyboard output:screen input:screen (forced read) output:screen
Disk	4 6 8 9 12	input read disk directory output, new file output,append input and output, update
Interface	5 8 9 13	concurrent input block output concurrent output concurrent input and output

aux2 is a device-specific parameter, and is usually set to 0. Generally, *aux2* is only used when opening the screen display for a graphics mode.

The final argument of an OPEN statement is the file specification. A file specification consists of a device and an optional filename (only with disk device). The device names used by the Atari are listed below. In the OPEN statement, the *filespec* must appear in quotation marks.

Cassette unit	C:
Editor	E:
Keyboard	K:
Printer	P:
Screen	S:
Disk	D:
Interface	R:

Cassette Unit

A *filenumber* can be opened for the cassette unit 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 proceding.

When an OPEN statement is executed for output to the cassette unit, the tone sounds twice. This is a reminder for the operator to press PLAY and RECORD on the cassette unit, followed by RETURN on the keyboard. For input, the tone sounds once to remind the operator to press PLAY on the cassette unit, followed by RETURN on the keyboard.

aux2 in an OPEN statement for the cassette unit can be assigned either 0 or 128. Files will be recorded with shorter gaps between the records when *aux2* = 128.

When an OPEN statement is executed, and the correct levers on the cassette unit are pressed, the cassette unit 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 the 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 will appear on the tape.

When a long section of blank tape (30 sec. or more) is encountered during input, an ERROR-138 (Device timeout) occurs. To avoid these errors, the device 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.

aux2 of an OPEN statement for the keyboard is ignored.

Example 1

```
10 DIM A$(1)
20 OPEN #2, 4, 0, "K:"
30 GRAPHICS 3 + 16
40 INPUT #2,A$
50 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 an 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

A *filenumber* can be opened for any of the disk I/O operations listed in table 8.3. 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 a filename and may include an optional filename extention. If included, the filename extention must be separated from the filename by a period.

The statements in example 2 are correct OPEN statements for a disk.

Example 2

```
OPEN #1, 4, 0, "D2:GRADES.BAS"
OPEN #3, 12, 0 "D:JONES"
```

Printer

An I/O channel for the printer may only be opened for output. The printer must be powered-up before an OPEN statement may be executed, and, if used with the Atari 850 interface, the interface must also be activated.

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 *aux1* of the OPEN statement (Table 8.3). 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 3

```
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
```

Example 3 contains a program that uses a display screen as an input device. Line 10 opens I/O filenumber 1 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 filenumber 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.

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 *filenumber*. *aux1* determines the I/O operation, as listed in Table 8.3. Although *aux2* 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.

Example 4 contains correct OPEN statements for the interface module.

Example 4

```
OPEN #1, 5, 0, "R2:"
OPEN #2, 13, 0, "R:"
OPEN #4, 8, 0, "R4:"
```

Screen

The OPEN statement for the screen device (S:) is used to configure the display. *aux1* selects whether the screen may be used for input. The screen device may always be used for output. Also, *aux1* determines if the display has a text window and if the display is cleared when the OPEN statement is executed *aux2* selects the graphics mode.

Table 8.4. Screen I/O operations

OPERATION NUMBER	OUTPUT	INPUT	Text WINDOW	Clear SCREEN
(aux 1)				
4 (12)	X	X		X
8	X		X	
20 (28)	X	X	X	X
24	X		X	X
36 (44)	X	X		
40	X			
52 (60)	X	X	X	
56	X		X	

Generally, when the screen is used with an OPEN statement instead of a GRAPHICS statement, the PLOT and DRAWTO comands cannot be used. Input is performed by the GET statement, while output is done with either PUT or PRINT#. Each of these statements requires a *filenumber* that matches to the *filenumber* of the OPEN statement. However, if *filenumber* = 6, both PLOT and DRAWTO will be functional regardless of how the screen was opened.

There are few exceptions to the rules given by Table 8.4. Graphics modes 0, 9, 10, and 11 may have no text window. Also, the screen will always clear when entering graphics mode 0.

Example 5

```
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
```

Example 5 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 the foreground color. Lines 30 and 40 draw a small diagonal line in the upper left of the display.

At line 50, the screen is opened as an I/O device. *aux2* = 60 indicates that the screen will be used for input and output, that a text window will be present and that the screen will not be cleared (see Table 8.4). *aux2* 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.

OR

Operator

OR is a logical operator. This reserved word is generally used to combine two comparisons in the context of an IF...THEN statement.

Configuration

expression1 OR *expression2*

If an *expression* is non-zero, that *expression* will be evaluated as true. Likewise, an *expression* with a value of zero will be evaluated as false. The following is the truth table for OR.

X	Y	X OR Y
true	true	true
true	false	true
false	true	true
false	false	false

In Atari BASIC, a true result is represented by a 1, and a false by 0.

Example

```
10 A = 3
20 B = 5
30 IF (B<A) OR (B=5) THEN 50
40 END
50 PRINT "EITHER B IS LESS THAN A"
60 PRINT "OR B IS EQUAL TO 5"
70 END
RUN
EITHER B IS LESS THAN A
OR B IS EQUAL TO 5
```

In the preceding example, B is not less than A, but B is equal to 5. Therefore, the whole OR expresson is true, and the program branches to line 50.

PADDLE

Function

The PADDLE function returns an integer between 1 and 228 that depends on the rotation of a particular paddle.

Configuration

PADDLE (*argument*)

A total of 4 paddle game controllers may be used at one time. The value of *argument* indicates the paddle number. If *argument* is not an integer, it will be rounded. The paddles are numbered 0 to 3; however PADDLE will accept *argument* in the range 0-255. If the PADDLE function has an *argument* in the range 4-255, the results are unpredicable. If a paddle is not present when the PADDLE function is executed, the value 228 is returned.

The paddle controllers are only used in pairs. A pair of controllers is plugged into one of the controllers jacks on the side of the computer. The first jack accepts paddles 0 and 1. The second jack accepts paddles 2 and 3.

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) is specified, the paddles must be plugged into controller jack 1.

PEEK — Function

The PEEK function is used to recover the value in a memory location.

Configuration

PEEK (*argument*)

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* is not an integer, it will be rounded off.

Many memory locations are of general interest. The contents of a memory location can be changed with a POKE statement. Appendix F contains information about commonly used memory locations.

Example

```
PRINT PEEK (83)
39
```

The previous example displays the current value of the right center margin screen. The default value is 39.

PLOT (PL.) — Statement

The PLOT statement is used to illuminate a character or picture element on the display. PLOT will output the data that has been selected by the last COLOR statement.

Configuration

PLOT *column, row*

column and *row* are numeric expressions that determine the position on the screen where the character or pixel will appear. The currently active graphics mode determines the allowable value for *column* and *row*. If either *row* or *column* is not an integer, it will be rounded. If either argument is negative or greater than the dimensions of the screen, an error will result.

Example

```
10 GRAPHICS 3
20 COLOR 2
30 FOR COL=0 TO 39 STEP 3
40 FOR ROW=0 TO 21 STEP 3
50 PLOT COL,ROW
60 NEXT ROW
70 NEXT COL
```

The previous example program illustrates the use of PLOT. Line 10 activates graphics mode 3; line 20 chooses color register 1. Since no SETCOLOR statement has been executed, color register 1 remains at its default value, green. Line 30 begins a FOR...NEXT loop that is executed 14 times. The value of its counter, COL, is successively set to 0, 3, 6, 9,...39. The inner loop, beginning at line 40, is executed once for every value of COL. During an execution of the inner loop, its counter, ROW, is successively set to 0, 3, 6, 9,...21. When all is said and done, the PLOT statement in line 50 will be executed 112 times. As a result, 112 pixels will be plotted in a grid-like pattern on the screen.

POINT (P.) Statement

The POINT command sets the location of the file pointer for a specified disk file. POINT is not available in version 1.0 of the disk operating system although it is supported in versions 2.0S and 2.5.

Configuration

POINT *#filenumber, variable1, variable2*

The POINT statement must specify a *filenumber* that is presently opened to a disk file.

With DOS 2.0S and DOS 2.5, *variable1* is a numeric variable that sets the sector number of the file pointer. *variable2* is also a numeric variable. It sets the byte number of the file pointer within the specified sector. Notice that both *variable1* and variable2 **must** be numeric variables. They may not be numeric constants or expressions.

Example

```
100  DIM A$ (10)
110  OPEN #2,8,0, "D:JOE"
120  NOTE #2, WHERE, DUMMY
130  PRINT #2, "LIVES HERE"
140  CLOSE #2
150  OPEN #2,4,0, "D:JOE"
160  DUMMY=DUMMY+10
170  POINT #2, WHERE, DUMMY
180  INPUT #2, A$
190  PRINT A$
200  CLOSE #2
RUN
LIVES HERE
```

POKE (POK.) — Statement

The POKE statement is used to store one byte of information in a particular memory location.

Configuration

POKE *address, value*

address specifies a 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. *value* represents one byte, and therefore, must be an integer between 0 and 255.

If either of the arguments of a POKE statement is not an integer, it will be rounded. A value error occurs if the *address* specified is greater than 65535 or the *value* exceeds 255. An 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 F 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

Statement

The POP statement causes a program to ignore the most recent GOSUB or ON...GOSUB statement. POP may also be used to prematurely exit a FOR...NEXT loop.

Configuration

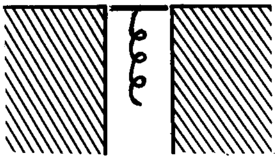
```
POP
```

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. POP deletes the top entry on the **run-time stack**.

Example

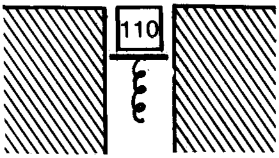
```
100 GOSUB 200
110 PRINT "PROGRAM FINISHED"
120 END
200 GOSUB 300
215 PRINT "MIDDLE ROUTINE"
230 RETURN
300 POP
310 PRINT "LAST ROUTINE"
320 RETURN
RUN
LAST ROUTINE
PROGRAM FINISHED
```

The run-time stack contains the return addresses from the subroutines. Before the program is executed the stack is cleared:



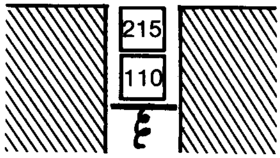
nothing on stack

The subroutine call in line 100 places the value of 110 on the stack. A subsequent RETURN statement would continue program execution at line 110.



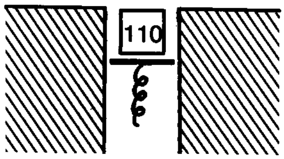
after "100 GOSUB 200"
is executed

From line 100, program execution continues with line 200. The GOSUB, here, places a 215 on the run-time stack.



after "200 GOSUB 300" is executed

From line 200, program execution continues with line 300. The POP, here, removes the top value from the stack.



after "300 POP" is executed

Line 310 prints the message "LAST ROUTINE", then line 320 executes a RETURN. The RETURN statement gets the top value from the stack, then resumes program execution at this line number. Therefore, line 110 is executed, printing the message, "PROGRAM FINISHED". Line 120 ends the program. Notice that the POP statement caused the program to forget the "MIDDLE ROUTINE".

Likewise, a POP statement can be used to make the program ignore the previous FOR statement. When POP is executed within a FOR... NEXT loop, that loop will not be replaced. However, an error will occur if a NEXT statement is executed for that loop. The correct way to exit a FOR...NEXT loop is illustrated in the following example.

Example

```
10 FOR I=1 TO 10
20 IF I^3>500 THEN POP: GOTO 50
30 NEXT I
40 END
50 PRINT "THE CUBE OF";I;" IS GREATER THAN 500"
```

POSITION (POS.) Statement

The POSITION statement moves the cursor to the specified *column* and *row*.

Configuration

POSITION *column,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, or INPUT 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* = 5 and *row* = 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 its left edge.

PRINT (PR. or ?) Statement

The PRINT statement is used to display data on the screen.

Configuration

PRINT [*expression*] [;] ...
? [*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 1

```
10 DIM A$(15)
20 A$ = "THOMAS R SMITH"
30 X = 27
40 PRINT "NAME:";A$,"AGE:";X
50 END
```

Example 1 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 contant "AGE:" and the value of the variable X are displayed in the last column.

Incidentally, the comma stops need not be set at intervals of 10 spaces. The memory location 201 contains the current comma stop width. POKE 201,20 would set the tab width to 20 spaces.

Example 2

```
10 POKE 201,15
20 PRINT,"15"
30 POKE 201,25
40 PRINT,"25"
```

PRINT# (PR.# or ?#) Statement

The PRINT# statement is used to output data to an I/O device.

Configuration

PRINT# *filenumber* [;][*expression*]...
?# *filenumber* [;][*expression*]...

filenumber indicates the I/O channel through which to output data. This *filenumber* must have been previously opened in the program. PRINT# operates in a manner similar to PRINT. The commas and semi-colons operate in an analogous fashion.

RESTORE

Statement

A RESTORE statement is used to move the data pointer.

Configuration

RESTORE [*linenumber*]

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.

If a RESTORE is executed without a *linenumber* being given, the next READ statement executed will read the first data item in the first DATA statement in the program. If a *linenumber* is given with the RESTORE statement, the next READ statement will read the first data item in the DATA statement named in *linenumber*.

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.)

Statement

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

```
10 GOSUB 100
20 PRINT "END"
30 END
100 PRINT "SUBROUTINE"
110 RETURN
RUN
SUBROUTINE
END
READY
```

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 — Function

The RND function is used to generate random numbers.

Configuration

RND (*argument*)

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(1) * 100)

The previous example contains a statement that generates random integers between 0 and 99 inclusive.

RUN (RU.) Statement

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 [*"filespec"*]

filespec consists of a device name and an optional filename. Disk files require a filename.

A RUN statement closes all files 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 cassette unit 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 cassette unit 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.

Example

```
RUN "C:"
RUN "D2:JONES.BAS"
```

SAVE — Statement

The SAVE command is used to send a BASIC program in RAM to an output device.

Configuration

SAVE "filespec"

filespec consists of a device name, such as the cassette unit (C:) or disk drive (D:), and and optional filename. In the case of the disk drive, the filename is required.

Files stored via SAVE are transferred in a tokenized format. These files can only be subsequently loaded using LOAD or RUN. ENTER will not load a program stored with SAVE.

Cassette Unit

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 transferrred from RAM to the cassette unit.

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.

Example

```
SAVE "D:GRIM"
SAVE "C:"
```

SETCOLOR (SE.) Statement

The SETCOLOR statement is used to change the default *color* and *luminance* of a specified color *register*.

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 6.2. The *luminance* can range from 0 (darkest) to 14 (brightest).

Color Register	Default Color
0	ORANGE
1	LIGHT GREEN
2	DARK BLUE
3	RED
4	BLACK

Example

```
100 GR.3+16
110 COLOR 1
120 FOR I = 1 TO 39 STEP 2
130 PLOT I,O:DRAWTOI,23
140 NEXT I
150 FOR I = 0 TO 15
160 FOR J = 0 TO 15
170 SET COLOR 0,I,J
180 NEXT J
190 NEXT I
```

SGN — Function

The SGN function returns a +1 if its *argument* is positive, a -1 if negative, and a 0 if zero.

Configuration

SGN (*argument*)

Example

```
100 A = 100
200 X = SGN (A)
300 PRINT X
RUN
1
```

SIN — Function

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 function.

Configuration

SIN (*argument*)

Example

```
10 DEG
20 X = SIN (90)
30 PRINT X
RUN
1
```

SOUND

Statement

The SOUND statement is used to output sound via the television set or monitor's speaker.

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, 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, and 15) 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 RESET key is pressed, sound will be turned off. However, if the BREAK key is pressed, sound will not be turned off.

SQR — Function

SQR returns the square root of its *argument*.

Configuration

SQR (*argument*)

Example

```
10 X = 49
20 PRINT SQR (X)
RUN
7
```

STATUS — Statement

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 8.5.

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 8.5. STATUS code values

STATUS Code	Reference
1	Operation completed with no problem.
3	Approaching end of file, Next READ receives last data in file.
128-171	Reference error messages 128-171 in appendix A.

STICK

Function

The STICK function returns the position of the joystick indicated as its *argument*.

Configuration

STICK (*argument*)

argument indicates the joystick number (0 or 1). The value returned can range from 0 to 15 and corresponds to the positions indicated in figure 8.1.

Example

IF STICK (1) = 7 THEN GOTO 700

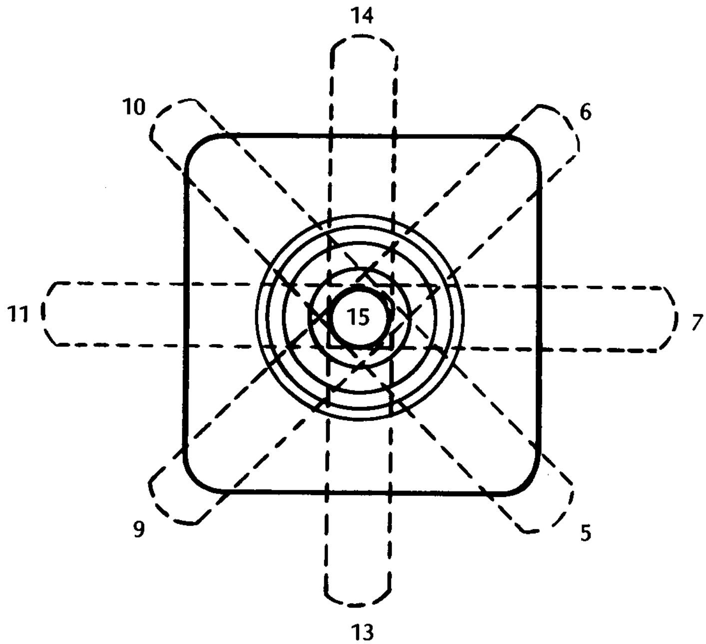


Figure 8.1. STICK Joystick Positions

STRIG

Function

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 (*argument*)

argument indicates the joystick number (0 or 1).

Example

```
100 IF STRIG (0) = 0 THEN GOTO 700
```

STOP

Statement

The STOP statement cause program execution to halt as though the BREAK key were pressed. (files are not closed, sound is not deactivated, etc.)

Configuration

STOP

If STOP is executed in the program mode, the following screen message will be displayed:

STOPPED AT LINE *number*

number is the line number where STOP was executed. If STOP is executed in the immediate mode, the following message will appear:

STOPPED

After program execution has been halted by STOP, it may be resumed using CONT.

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$ — Function

STR$ returns the string representation of its numeric *argument*.

Configuration

STR$ (*argument*)

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) could not be used in calculations.

Example

```
050 DIM A$(5)
150 A$ = STR$(40)
200 PRINT A$, LEN(A$)
RUN
40          2
```

TRAP — Statement

The TRAP statement causes program execution to branch to the *linenumber* indicated when an error is encountered.

Configuration

TRAP *linenumber*

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. The error handling routine can be reactivated with the following statement:

```
TRAP 40000
```

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, returns the line number where the error occurred:

```
256 * PEEK(187) + PEEK(186)
```

In our example, the data input in response to the INPUT statement in line 200 was string data. Since a numeric variable was specified in line 200, ERROR-8 (INPUT statement error) was generated. This was displayed along with the line number where the error occurred (200).

USR Function

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 figure 8.2

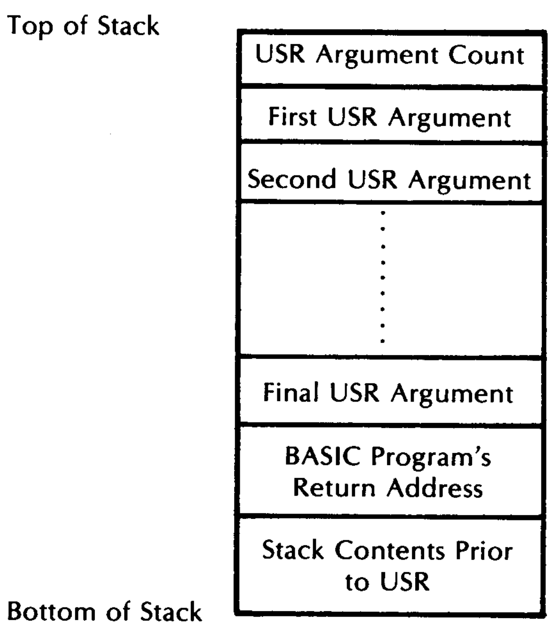


Figure 8.2. USR Hardware Stack

Returning to BASIC

When BASIC executes a USR function, the BASIC program's current location is pushed onto the hardware stack (see figure 8.2). 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 langauge program must pull the argument count off the stack before returning to the BASIC program

Example

X = USR (58487)

The preceding example will boot the Atari as if the computer had been just powered-up. Anything contained in memory will be lost.

VAL

Function

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 (*argument*)

Example

```
 50 DIM A$(50)
100 A$ = "57A72B"
200 PRINT VAL(A$)
300 PRINT VAL(A$) + 2
RUN
57
59
```

XIO

Statement

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 8.6.

Configuration

XIO *command,# filenumber,aux1,aux2,aux3*

The *command* value (as specified in table 8.6) indicates the operation to be performed. Generally, the *filenumber* specified must have been previously opened for input or output.

The auxillary expressions (*aux1, aux2, aux3*) are not always used by XIO, however, they must always be present as parameters. Generally, *aux3* specifies the device to be used for the input/output operation.

Example

```
100 GRAPHICS 15
110 COLOR 2
120 PLOT 80,80
130 DRAWTO 140,20
140 DRAWTO 19,20
150 POSITION 79,80
160 POKE 765,3
170 XIO 18, #6, 0, 0, "S:"
```

The preceding example illustrates the use of the XIO statement to fill an area in graphics. *command* = 18 specifies the graphics fill-area action. 6 is the graphics *filenumber*. The numeric parameters are both specified as 0, and the device is the screen (*aux3* = "S:")

Table 8.6. XIO Command Summary

Operation	Command	Equivalent Command	*aux 1*	*aux 2*	*aux 3*
General I/O Operations:					
Open a channel	3	OPEN	identical to OPEN statement parameters		
Read a line	5	INPUT#	4	0	dimensioned string variable
Read 255 characters	7	—	4	0	dimensioned string variable
Write a line	9	PRINT#	8	0	string data
Write 255 characters	11	—	8	0	string data
Close channel	12	CLOSE	0	0	string
Status of channel	13	STATUS	0	0	string
Screen Graphics:					
Draw a line	17	DRAWTO	0	0	"S:"
Fill an area	18	—	0	0	"S:"
Disk:					
Rename	32	DOS 2 Menu E	0	0	"D:*old,new*"
Delete	33	DOS 2 Menu D	0	0	"D:*file*"
Lock	35	DOS 2 Menu F	0	0	"D:*file*"
Unlock	36	DOS 2 Menu G	0	0	"D:*file*"
Move file pointer	37	POINT	0	0	—
Fine file pointer	38	NOTE	0	0	—
Directory entry	39	INPUT#	0	0	string variable
Wildcard decipher	40	—	0	0	—
Format single-density	253	DOS 2	33	87	"D:"
	254	Menu I	0	0	"D:"
Format dual-density	253	—	33	127	"D:"
RS232 Port: (Interface module)					
Force short block	32	—	0	0	"R:"
Control DTR,RTS,XMT	34	—	table 8.7	0	"R:"
Baud rate, word size, stop bits, and ready monitoring	36	—	table 8.9	table 8.10	"R:"
Translation mode	38	—	table 8.8	ASCII code	"R:"
Concurrent mode	40	—	0	0	"R:"

Table 8.7. *aux1* values for RS232 Port

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 values for DTR, RTS, & XMT to obtain *aux1*

Example Values of *aux1*	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 8.8. *aux1* values 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	0	Light	0	Disregard	0	No change	0
Yes**	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 *aux1*

** The line feed character is appended after a carriage return (EOL).

Table 8.9. *aux1* 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 *aux1*

Table 8.10. *aux2* values for XIO 36
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

9

Advanced Memory Concepts

In this chapter the means of accessing the full 128K of memory in the Atari 130XE will be covered in detail. In addition, the general concepts of the memory and operating system in the Atari 130XE will be discussed. Although the explanation will center around the memory system in the 130XE, the concepts explained in these sections will be extremely helpful to any computer operator.

130XE RAM System

The Atari 130XE contains 131,072 bytes (128K) of Random Access Memory (RAM), twice the amount contained in the Atari 65XE or 800XL. Because the computer is designed around the 6502C microprocessor, it cannot, however, access more than 65536 bytes (64K) of that RAM at any one time (*see* figure 9.1).

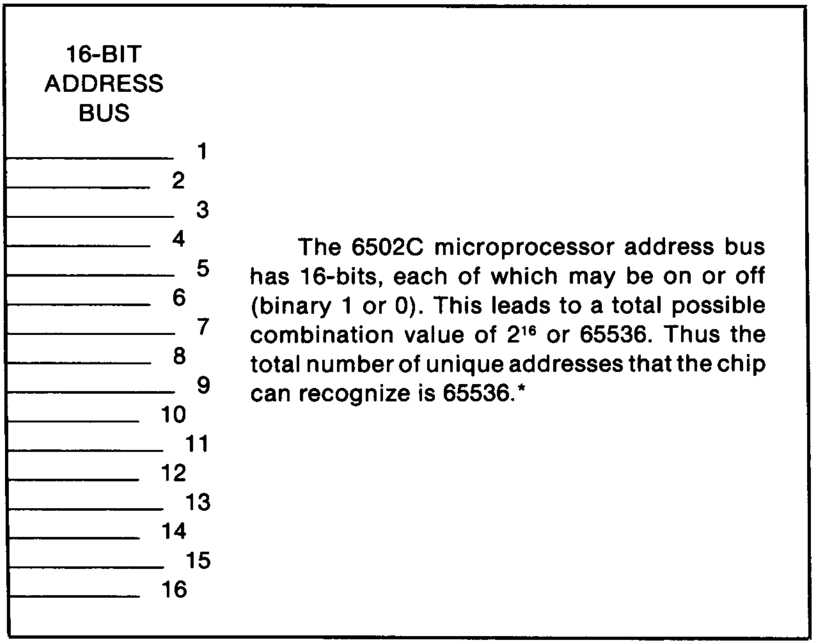


Figure 9.1. 16-bit address bus

To make up for this limitation of the 6502C, the Atari 130XE depends upon "bank switching" to utilize the extra 64K of memory. In this process, the computer accesses the extra memory by switching "out" one section of memory and switching "in" another section. While the computer never has access to more than 64K at any one time, it can use this method to store and retrieve information from all of the 128K of available memory.

* The number 2^{16}, which represents the total combinations of 1 and 0, is arrived at through application of basic probability theory.

The "bank switching" technique, in the Atari 130XE, operates on the section of RAM located between 16384 and 32767. The determination as to which bank of memory will occupy that region of memory depends on the setting of the "bank switch" located at memory location 54017. This location is used as the port B address of the 6520 Peripheral Interface Adaptor chip, which controls the computer input and output. By changing the value in the bank switch, any desired memory bank may be used. (Although no physical exchange occurs, it appears to the user that the memory sections have been switched.)

In dealing directly with the bank switch (which is actually a set of switches) it is necessary to understand how the bits of data in a byte of data are related. The purpose of the following section is to familiarize the user with the binary system. The applications of the binary system to the memory bank switching techniques, as well as to the general operation of the computer, will be discussed later in this chapter.

Binary System

To fully understand the concept of the binary system it is helpful to compare it to the decimal system. In the decimal (base 10) system values are represented using the digits 0-9. The digits are placed side by side with each digit being ten times the value of the digit preceding it (right to left). In the following example each two has 10 times the value of the preceding two (right to left):

222 = 2*100+2*10+2

The binary system is very similar to the decimal system in that the digits are placed side by side and increase in value moving from right to left. However, in the binary system (base 2) only the digits 1 and 0 are used. Also, moving from right to left, each digit has twice the value of the preceding digit. *See* figure 9.2.

Binary Digits

101010 = 1*(32)+0*(16)+1*(8)+0*(4)+1*(2)+0*(1)

Decimal Digits

101010 = $1*(10^5)+0*(10^4)+1*(10^3)+0*(10^2)+1*(10)+(0)$

Value Placement

Digit #	**...**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**...**
BINARY	...	128	64	32	16	8	4	2	1	...
DECIMAL	...	10^7	10^6	10^5	10^4	10^3	10^2	10	1	...

Note: ... — denotes that the values continue on infinitely in either direction. For the purpose of computer application, however, only this range will be discussed.

Figure 9.2. Binary -v- decimal

Conversion

In some instances programmers or users find it necessary to convert between the decimal and binary systems. The following sections will discuss the conversion process completely.*

BINARY TO DECIMAL

Converting a value written in binary form to decimal form is a relatively simple matter. It requires the addition of the place values corresponding to the columns where the ones digits are located. (In binary as in decimal, zeros are place holders only).

For example, in the binary number 1010, the ones are located in the second place (having a value of 2) and the fourth place (having a value of 8). Thus, the decimal value of the binary number 1010 is 10.

* Many programmers use an intermediate system known as hexadecimal (base 16). For the purpose of this text, however, a discussion of the hexadecimal system is unnecessary.

	Decimal	Binary						
		64	32	16	8	4	2	1
Example (1):	126	0	0	0	0	0	0	0
	126 - $\boxed{64}$	$\boxed{1}$	0	0	0	0	0	0
	62 - $\boxed{32}$	1	$\boxed{1}$	0	0	0	0	0
	30 - $\boxed{16}$	1	1	$\boxed{1}$	0	0	0	0
	14 - $\boxed{8}$	1	1	1	$\boxed{1}$	0	0	0
	6 - $\boxed{4}$	1	1	1	1	$\boxed{1}$	0	0
	2 - $\boxed{2}$	1	1	1	1	1	$\boxed{1}$	0
	0							

	Decimal								
		128	64	32	16	8	4	2	1
Example (2):	135	0	0	0	0	0	0	0	0
	135 - $\boxed{128}$	$\boxed{1}$	0	0	0	0	0	0	0
	7 - $\boxed{64}$ = -57	1	$\boxed{0}$	0	0	0	0	0	0
	7 - $\boxed{32}$ = -25	1	0	$\boxed{0}$	0	0	0	0	0
	7 - $\boxed{16}$ = -9	1	0	0	$\boxed{0}$	0	0	0	0
	7 - $\boxed{8}$ = -1	1	0	0	0	$\boxed{0}$	0	0	0
	7 - $\boxed{4}$	1	0	0	0	0	$\boxed{1}$	0	0
	3 - $\boxed{2}$	1	0	0	0	0	1	$\boxed{1}$	0
	1 - $\boxed{1}$	1	0	0	0	0	1	1	$\boxed{1}$
	0								

Figure 9.3. Decimal to binary

DECIMAL TO BINARY

The process by which a decimal number is converted to binary form is somewhat more difficult than the binary to decimal conversion. This is because the decimal number must be broken down to fit into the binary format. To convert the decimal value, the first step is to find the binary column which has the largest value that is less than the decimal value. Then, a one is placed in that column and its value is subtracted from the decimal number. This process is repeated with the remainder, after subtracting the column's value, until the remainder is zero. *See* figure 9.3

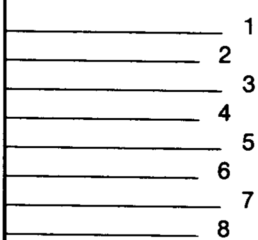
example (1). If while converting, the subtraction results in a negative value, then a zero should be placed in that binary column and it should be skipped. (Proceed to the next column; do **not** subtract the value). *See* figure 9.3 example (2).

The Computer and The Binary System

The basic unit that a computer operates with is known as a bit or "binary digit". As in inflected by the term, "binary digit", the computer operates in the binary system. The computer can only recognize a one or a zero.* Using the information covered in the previous section designers have devised a system by which the computer, using combinations of ones and zeros, can represent more complex values.

In most microcomputers, including the Atari 130XE, more complex units such as alphanumeric characters are represented using 8-bit sets known as bytes. These complex codes are used to display, calculate, and transmit data. However, because microcomputers are designed to handle the 8-bit form of data, they are limited to an 8-bit data bus. This means that no byte of data can have a value greater than 255. *See* figure 9.4.

8-bit Data bus



2^8=256

Since the data bus is set up for 8-bits (byte), it can only handle a combination of 256 values (0-255).

Lowest Value

00000000 = 0(128)+0(64)+0(32)+0(16)+0(8)+0(4)+0(2)+0=0

Highest Value

11111111=1(128)+1(64)+1(32)+1(8)+1(4)+1(2)+1=255

Figure 9.4. 8-bit data bus

* Computers actually deal with voltage (1) or no voltage (0).

Bit#	7	6	5	4	3	2	1	0
Decimal Value	128	64	32	16	8	4	2	1
State 1 or 0	1	1	A	B	C	D	0	1

Figure 9.5. Location 54017

Bit A	Bit B	Antic access	C.P.U. access
0	1	extra	normal
1	0	normal	extra
1	1	normal	normal
0	0	extra	extra

Extra: refers to the extra bank of memory that bits C and D determine

Normal: refers to normal access memory. In this mode C and D have no effect on this device.

Figure 9.6. Processor access

Bit-Controlled Operations

The computer itself operates on the binary level at all times. The complex characters, created by the bytes of data are purely for the user's comprehension. Because of this it is sometimes difficult for a user to comprehend the purpose of placing certain values in specific places to accomplish a desired task. This is because the user is examining the data he is using on his level or at the 8-bit level. However, the computer may be dealing with a single bit, and the result in a confused operator.

The following section will address this question by providing a thorough explanation of an example of just a procedure. Although the concepts in the following example can be applied to many operations, it is designed specifically to clarify "bank switching".

Bit Control Of The "Bank Switch"

Earlier in this chapter, the significance of memory location 54017 in the Atari 130XE, the bank switch, was discussed briefly. The location is used to direct input and output to and from the central processing unit and/or the Antic (graphics) microprocessor. This section is designed to teach the user to alter bits of data and thus gain better control over the bank switch.

The control of the bank switch depends on four bits (a nibble) of location 54017. In figure 9.5 these bits are labeled A, B, C, and D.

Bits A and B determine which device (Antic and/or CPU) has access to the extra memory. Since there are only two bits controlling the access, there can only be four states of access. *See* figure 9.6.

Bits C and D control which bank of memory the device specified by bits A and B can access. Because they only determine which section of extra memory the device can access, bits C and D have no effect if A and B are "set" to one (when normal memory is being accessed). It is for this reason that different values can be "poked" into the location 54017 without altering its effect *see* figure 9.7.

```
253=128+64+32+16+8+4+0+1
      1   1   1   1  1  1 0 1
                  A   B  C D
241=128+64+32+16+0+0+0+1
      1   1   1   1  0  0 0 1
```

Since bits A, B are "set", access is to normal memory and C D have no effect. Therefore, if either value was poked into location 54017, the CPU and Antic would be directed to access normal memory.

Figure 9.7. Bits C and D

As was previously stated, bits C and D in figure 9.5 determine which section of extra memory will be accessed. Again, since only two bits are controlling the bank selection, only four possible banks can be selected. *See* figure 9.8. Although some texts refer to these banks as being sequentially oriented (i.e. bank #1: 0 to 16383, bank #2: 16384 to 32767), it is not necessary to do so. This is because each bank can only be accessed individually. In other words, the CPU and the Antic chips may both be directed to access the same memory (normal or extra) or they may be split between the normal memory and one extra memory bank. Thus it is impossible to access more than one extra memory bank at a time.

Combination C	D	"Extra" Memory Access
0	0	bank #1
0	1	bank #2
1	0	bank #3
1	1	bank #4

Figure 9.8. Memory Access

To control the access, all the programmer need do is develop a binary byte which fits the task he requires, convert it to decimal and poke that value into location 54017. Remember, however, that in figure 9.5 the bits numbered (0,6,7) are shown to be "set" (ones) and bit 2 is not "set" (zero). These bits must always be in this state or the computer will lose track of input and output and "lockup".

Note: For a complete listing of the numbers that may be poked into location 54017 and their results, refer to appendix G.

Using "Bank Switching" in BASIC

The following program effectively illustrates the memory bank switching process. It should clarify the methods described in the previous chapter.

When the program runs, five messages are written to the five banks of memory used in the process (normal memory and the four (16K) extra banks). The program then reads back the messages to verify that the extra banks were used.

The important loop in the program occurs between line 150 and 230. Within this loop, line 190 changes the bank in which data will be written. The sequence through the loop as it affects line 190 is as follows:

1. When x=0 ⇨ POKE 54017,221
2. When x=1 ⇨ POKE 54017,225
3. When x=2 ⇨ POKE 54017,229
4. When x=3 ⇨ POKE 54017,233
5. When x=4 ⇨ POKE 54017,237

By referring to table G.1 in appendix G, it is evident that as x and the 54017 "poked" values are incremented (x by one and the poked value by multiples of four), the memory bank switch (CPU) is set first to normal, and then to each bank. The bit changes that correspond to the decimal changes of the "poked" value can be seen in table G.2 of appendix G.

The process for reading the data is exactly the same only the data is read not written.

```
100 REM BANK SWITCHING
110 REM By Ron Lee
120 REM
130 DIM A$(30)
140 PRINT CHR$(125)
150 FOR X=0 TO 4
160 IF X=0 THEN PRINT "WRITING TO NORMAL MEMORY"
170 IF X<>0 THEN PRINT "WRITING TO BANK #";X
180 READ A$
190 POKE 54017,221+X*4
200 FOR Y=1 TO LEN(A$)
210 POKE 16384+Y,ASC(A$(Y,Y)):PRINT A$(Y,Y);
220 NEXT Y:PRINT:PRINT
230 NEXT X
240 RESTORE
250 GOSUB 600:PRINT CHR$(125)
260 FOR X=0 TO 4
270 IF X=0 THEN PRINT "READING FROM NORMAL MEMORY"
280 IF X<>0 THEN PRINT "READING FROM BANK #";X
290 READ A$
300 POKE 54017,221+X*4
310 FOR Y=1 TO LEN(A$)
320 PRINT CHR$ (PEEK(16384+Y));
330 NEXT Y
340 PRINT:PRINT
350 NEXT X
360 END
500 DATA TIFFANY WAS HERE, BEAUFORD WAS HERE,
    RON WAS HERE
510 DATA NUKE WAS HERE, SAM WAS HERE
600 POSITION 2,22:PRINT "Press <RETURN> to Continue"
610 OPEN #5,4,0,"K:"
620 GET #5, A:IF A=155 THEN CLOSE #5: RETURN
630 GOTO 620
```

Appendix A.
Atari Error Messages

This appendix describes the error numbers used by the Atari. Error numbers less than 128 are application specific. That is, the meaning of each code depends on whether BASIC or DOS is active. Error numbers greater than 127 generally result from an I/O error and keep their meaning regardless of the application.

Error #	Error Name	Cause
2 (BASIC)	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 a GOSUB statement with too many levels of nesting.
2 (DOS)	No command file found	The "X-user-defined" option of the DOS 3 menu was attempted, but no files of the form *.CMD were contained on drive #1.
3 (BASIC)	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.
3 (DOS)	Input required	Only the RETURN key was pressed in response to a prompt that required an input.
4 (BASIC)	Too many variables	Over 128 variable names have been specified. Any unused names should be deleted by executing the following lines. L."D:TEMP" NEW E."D:TEMP" The cassette unit could also be used to delete the names in a similar manner.

Error #	Error Name	Cause
4 (DOS)	No cartridge	The "To cartridge" Menu Option of DOS 3 was attempted; however no cartridge was present and BASIC had been deactivated.
5 (BASIC)	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.
5 (DOS)	I/O error	A generic input/output error.
6 (BASIC)	Out of data error	The DATA statements did not contain enough data items for the variables in the corresponding READ statements.
6 (DOS)	Invalid end address	The End address for the "Save" option was entered as less than the Start address.
7 (BASIC)	Line number greater than 32767	The line number is negative or greater than 32767.
7 (DOS)	Error loading MEM.SAV	The Atari has not been able to reload the RAM using MEM.SAV. Possible causes include a faulty disk or a dirty drive.
8 (BASIC)	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.
8 (DOS)	Error saving MEM.SAV	The MEM.SAV file on disk is no longer valid after this error.
9 (BASIC)	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. Or if an attempt is made to DIM a string of length zero or length greater than 32767.

Error #	Error Name	Cause
9 (DOS)	Drive input error	An invalid device specification was supplied.
10 (BASIC)	Argument stack Overflow	To many nested parenthesis in an expression.
10 (DOS)	Filename input error	An invalid filename was supplied.
11 (BASIC)	Floating point overflow/underflow	The program encountered a number with an absolute value less than 1E-99 or greater than 1E+98. This error also occurs when an attempt is made to divide by zero.
12 (BASIC)	Line not found	An IF-THEN, ON-GOSUB, ON-GOTO, GOSUB, or GOTO statement referenced a line number that does not exist.
13 (BASIC)	No matching FOR	A NEXT statement was encountered that did not have a corresponding FOR statement.
14 (BASIC)	Line too long	The line entered is greater than the length of the BASIC line processing buffer length.
15 (BASIC)	GOSUB or FOR line deleted	A NEXT statement was encountered for which the corresponding FOR or GOSUB statement had been deleted.
16 (BASIC)	RETURN error	A RETURN statement was encountered without a corresponding GOSUB statement.
17 (BASIC)	Garbage error	This error can be caused by faulty RAM or the incorrect use of a POKE statement.
18 (BASIC)	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 (BASIC)	LOAD program too long	The program being loaded will not fit in the available RAM.
20 (BASIC)	Device number error	A device number outside of the range 0 to 7 was entered.

Error #	Error Name	Cause
21 (BASIC)	LOAD file error	The LOAD statement was incorrectly used to load a program that was not stored using the SAVE format.
128	BREAK abort	The BREAK key was pressed during an I/O operation causing execution to stop.
129	ICOB* already open	This error occurs when an attempt is made to use a filenumber currently in use. Often, the filenumber 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").
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 filenumber 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 filenumbers 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.

* IOCB — Input/output control block

Error #	Error Name	Cause
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 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 device 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 of 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).
147	Insufficient RAM	More RAM is require for the graphics mode chosen. Either add RAM or change graphics modes.

Error #	Error Name	Cause
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 the 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 4 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.
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, rename, or erase a locked file.

Error #	Error Name	Cause
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 in DOS 2.0S (63 in DOS 3).
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 1.0 file for append using the DOS 2.0S operating system. Try copying the DOS 1.0 file to a DOS 2.0S diskette using DOS 2.0S. It is illegal for DOS 2.0S to append to DOS 1.0 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.
174	Duplicate filename	A "Rename" has been attempted that would have resulted in two files of the same name on a diskette.
175	Bad load file	The specified file is not a load-type file.
176	Incompatible format	This error occurs when a DOS 3 operation is attempted using a DOS 2.0S diskette. Use "Access DOS 2" to translate the file to rectify the situation.
177	Disk structure damaged	DOS 3 does not recognize the file on the disk, due to damage on the disk. (May be caused from use of a non-DOS 3 diskette).

Appendix B.
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 codes produce control characters. When control characters are output using a PRINT statement, nothing is actually displayed on the screen. Instead 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 using an **escape sequence** enclosed within quotation marks.

To produce an escape sequence, first press the ESC key, and then press the keys which will produce the desired control character. For example, if the ESC key is pressed prior to pressing the CONTROL key and the = key, the Atari 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                    ESC then CONTROL- =
PRINT "NNN↓A"            pressed here
```

The output produced would be;

```
NNN
   A
```

Notice that when the ESC \ CONTROL- = keyboard entry was made, the control process specified (cursor down) did not actually occur. However, the screen character for cursor down (↓) was displayed on the screen.

When the PRINT statement was subsequently executed, the cursor down control process did take place. The result of this control process was the movement of the cursor one row down. This caused the "A" to be printed on the line below the line on which the three N's were output.

If the Atari 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,

```
                 ESC   ESC pressed here
PRINT "NN␛↓A"
                 ESC   CONTROL-= pressed here
```

the following output would be displayed on the screen;

```
NNN↓A
```

Notice that although the control process did not occur, the control character was displayed.

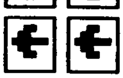
A great number of the Atari characters can only be entered via the keyboard when the keyboard is in the lowercase mode. By pressing the CAPS key once, the keyboard will be placed into the lowercase mode. Repressing the CAPS key will return the keyboard to the uppercase mode.

The XE series has two built-in character sets—the standard set and the extended set. For the majority of the characters these sets coincide. However, for a few of the ASCII codes the extended set will produce a different character than does the standard set. The standard set is selected if location 756 is assigned a value of 224, the extended set is selected if location 756 is assigned 204.

```
POKE 756,204     select extended
POKE 756,224     select standard
```

Most of the ASCII codes greater than 127 can be generated in the inverse video mode. Pressing the ◪ key toggles this mode. The symbol for this key will be listed with the key combination for the ASCII codes that require this mode to be active.

Atari ASCII Character Std. Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
	NULL	0	CONTROL-,
	SOH	1	CONTROL-A
	STX	2	CONTROL-B
	ETX	3	CONTROL-C
	EOT	4	CONTROL-D
	ENQ	5	CONTROL-E
	ACK	6	CONTROL-F
	BEL	7	CONTROL-G
	BS	8	CONTROL-H
	HT	9	CONTROL-I
	LF	10	CONTROL-J
	VT	11	CONTROL-K
	FF	12	CONTROL-L
	CR	13	CONTROL-M
	SO	14	CONTROL-N
	SI	15	CONTROL-O
	DLE	16	CONTROL-P
	DC1	17	CONTROL-Q
	DC2	18	CONTROL-R

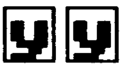
Atari ASCII Character Std. Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
	DC3	9	CONTROL-S
	DC4	20	CONTROL-T
	NAK	21	CONTROL-U
	SYN	22	CONTROL-V
	ETB	23	CONTROL-W
	CAN	24	CONTROL-X
	EM	25	CONTROL-Y
	SUB	26	CONTROL-Z
	ESC	27	ESC/ESC
	FS	28	ESC/CONTROL--
	GS	29	ESC/CONTROL-=
	RS	30	ESC/CONTROL-+
	US	31	ESC/CONTROL-*
	Space	32	SPACE BAR
	!	33	SHIFT-1
	"	34	SHIFT-2
	#	35	SHIFT-3
	$	36	SHIFT-4
	%	37	SHIFT-5

Atari ASCII Character Std.	Atari ASCII Character Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
&	&	&	38	SHIFT-6
'	'	'	39	SHIFT-7
(	(	(	40	SHIFT-9
)	)	)	41	SHIFT-0
*	*	*	42	*
+	+	+	43	+
,	,	,	44	,
-	-	-	45	-
.	.	.	46	.
/	/	/	47	/
0	0	0	48	0
1	1	1	49	1
2	2	2	50	2
3	3	3	51	3
4	4	4	52	4
5	5	5	53	5
6	6	6	54	6
7	7	7	55	7
8	8	8	56	8

Atari ASCII Character Std.	Atari ASCII Character Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
9	9	9	57	9
:	:	:	58	SHIFT-;
;	;	;	59	;
<	<	<	60	<
=	=	=	61	=
>	>	>	62	>
?	?	?	63	SHIFT-/
@	@	@	64	SHIFT-8
A	A	A	65	(SHIFT OR CAPS ON) A
B	B	B	66	(SHIFT OR CAPS ON) B
C	C	C	67	(SHIFT OR CAPS ON) C
D	D	D	68	(SHIFT OR CAPS ON) D
E	E	E	69	(SHIFT OR CAPS ON) E
F	F	F	70	(SHIFT OR CAPS ON) F
G	G	G	71	(SHIFT OR CAPS ON) G
H	H	H	72	(SHIFT OR CAPS ON) H
I	I	I	73	(SHIFT OR CAPS ON) I
J	J	J	74	(SHIFT OR CAPS ON) J
K	K	K	75	(SHIFT OR CAPS ON) K

Atari ASCII Character Std.	Atari ASCII Character Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
L	L	L	76	(SHIFT OR CAPS ON) L
M	M	M	77	(SHIFT OR CAPS ON) M
N	N	N	78	(SHIFT OR CAPS ON) N
O	O	O	79	(SHIFT OR CAPS ON) O
P	P	P	80	(SHIFT OR CAPS ON) P
Q	Q	Q	81	(SHIFT OR CAPS ON) Q
R	R	R	82	(SHIFT OR CAPS ON) R
S	S	S	83	(SHIFT OR CAPS ON) S
T	T	T	84	(SHIFT OR CAPS ON) T
U	U	U	85	(SHIFT OR CAPS ON) U
V	V	V	86	(SHIFT OR CAPS ON) V
W	W	W	87	(SHIFT OR CAPS ON) W
X	X	X	88	(SHIFT OR CAPS ON) X
Y	Y	Y	89	(SHIFT OR CAPS ON) Y
Z	Z	Z	90	(SHIFT OR CAPS ON) Z
[	[	[	91	SHIFT-;
\	\		92	SHIFT-,
]	]	]	93	SHIFT-+
^	^	↑	94	SHIFT-*

Atari ASCII Character Std.	Atari ASCII Character Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
_	_	_	95	SHIFT--
♦	i		96	CTRL-
a	a	a	97	(CAPS OFF) A
b	b	b	98	(CAPS OFF) B
c	c	c	99	(CAPS OFF) C
d	d	d	100	(CAPS OFF) D
e	e	e	101	(CAPS OFF) E
f	f	f	102	(CAPS OFF) F
g	g	g	103	(CAPS OFF) G
h	h	h	104	(CAPS OFF) H
i	i	i	105	(CAPS OFF) I
j	j	j	106	(CAPS OFF) J
k	k	k	107	(CAPS OFF) K
l	l	l	108	(CAPS OFF) L
m	m	m	109	(CAPS OFF) M
n	n	n	110	(CAPS OFF) N
o	o	o	111	(CAPS OFF) O
p	p	p	112	(CAPS OFF) P
q	q	q	113	(CAPS OFF) Q

Atari ASCII Character Std. Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
	r	114	(CAPS OFF) R
	s	115	(CAPS OFF) S
	t	116	(CAPS OFF) T
	u	117	(CAPS OFF) U
	v	118	(CAPS OFF) V
	w	119	(CAPS OFF) W
	x	120	(CAPS OFF) X
	y	121	(CAPS OFF) Y
	z	122	(CAPS OFF) Z
	{	123	CTRL-;
	|	124	SHIFT-=
	}	125	ESC/CTRL-<
			ESC/SHIFT-<
		126	ESC/BACK S
	DEL	127	ESC/TAB
		128	(◪) CONTROL-,
		129	(◪) CONTROL-A
		130	(◪) CONTROL-B
		131	(◪) CONTROL-C

Atari ASCII Character Std. Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
		132	(◪) CONTROL-D
		133	(◪) CONTROL-E
		134	(◪) CONTROL-F
		135	(◪) CONTROL-G
		136	(◪) CONTROL-H
		137	(◪) CONTROL-I
		138	(◪) CONTROL-J
		139	(◪) CONTROL-K
		140	(◪) CONTROL-L
		141	(◪) CONTROL-M
		142	(◪) CONTROL-N
		143	(◪) CONTROL-O
		144	(◪) CONTROL-P
		145	(◪) CONTROL-Q
		146	(◪) CONTROL-R
		147	(◪) CONTROL-S
		148	(◪) CONTROL-T
		149	(◪) CONTROL-U
		150	(◪) CONTROL-V

Atari ASCII Character Std. Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
		151	(◪) CONTROL-W
		152	(◪) CONTROL-X
		153	(◪) CONTROL-Y
		154	(◪) CONTROL-Z
		155	RETURN
		156	ESC/SHIFT- BACK S
		157	ESC/SHIFT->
		158	ESC/CTRL- TAB
		159	ESC/SHIFT- TAB
		160	(◪) SPACE BAR
		161	(◪) SHIFT-1
		162	(◪) SHIFT-2
		163	(◪) SHIFT-3
		164	(◪) SHIFT-4
		165	(◪) SHIFT-5
		166	(◪) SHIFT-6

Atari ASCII Character Std.	Atari ASCII Character Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
'	'		167	(◪) SHIFT-7
(	(		168	(◪) SHIFT-9
)	)		169	(◪) SHIFT-0
*	*		170	(◪) *
+	+		171	(◪) +
,	,		172	(◪) ,
-	-		173	(◪) -
.	.		174	(◪) .
/	/		175	(◪) /
0	0		176	(◪) 0
1	1		177	(◪) 1
2	2		178	(◪) 2
3	3		179	(◪) 3
4	4		180	(◪) 4
5	5		181	(◪) 5
6	6		182	(◪) 6
7	7		183	(◪) 7
8	8		184	(◪) 8
9	9		185	(◪) 9

Atari ASCII Character Std.	Atari ASCII Character Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
:	:		186	(◪) SHIFT-;
;	;		187	(◪) ;
<	<		188	(◪) <
=	=		189	(◪) =
>	>		190	(◪) >
?	?		191	(◪) SHIFT-/
@	@		192	(◪) SHIFT 8
A	A		193	(◪) (SHIFT OR CAPS ON) A
B	B		194	(◪) (SHIFT OR CAPS ON) B
C	C		195	(◪) (SHIFT OR CAPS ON) C
D	D		196	(◪) (SHIFT OR CAPS ON) D
E	E		197	(◪) (SHIFT OR CAPS ON) E
F	F		198	(◪) (SHIFT OR CAPS ON) F
G	G		199	(◪) (SHIFT OR CAPS ON) G
H	H		200	(◪) (SHIFT OR CAPS ON) H
I	I		201	(◪) (SHIFT OR CAPS ON) I
J	J		202	(◪) (SHIFT OR CAPS ON) J
K	K		203	(◪) (SHIFT OR CAPS ON) K
L	L		204	(◪) (SHIFT OR CAPS ON) L

Atari ASCII Character Std. Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
		205	(◩) (SHIFT OR CAPS ON) M
		206	(◩) (SHIFT OR CAPS ON) N
		207	(◩) (SHIFT OR CAPS ON) O
		208	(◩) (SHIFT OR CAPS ON) P
		209	(◩) (SHIFT OR CAPS ON) Q
		210	(◩) (SHIFT OR CAPS ON) R
		211	(◩) (SHIFT OR CAPS ON) S
		212	(◩) (SHIFT OR CAPS ON) T
		213	(◩) (SHIFT OR CAPS ON) U
		214	(◩) (SHIFT OR CAPS ON) V
		215	(◩) (SHIFT OR CAPS ON) W
		216	(◩) (SHIFT OR CAPS ON) X
		217	(◩) (SHIFT OR CAPS ON) Y
		218	(◩) (SHIFT OR CAPS ON) Z
		219	(◩) SHIFT-,
		220	(◩) SHIFT-+
		221	(◩) SHIFT-.
		222	(◩) SHIFT-*
		223	(◩) SHIFT--

Atari ASCII Character Std.	Atari ASCII Character Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
◆	i		224	(◪) CTRL-.
a	a		225	(◪) (CAPS OFF) A
b	b		226	(◪) (CAPS OFF) B
c	c		227	(◪) (CAPS OFF) C
d	d		228	(◪) (CAPS OFF) D
e	e		229	(◪) (CAPS OFF) E
f	f		230	(◪) (CAPS OFF) F
g	g		231	(◪) (CAPS OFF) G
h	h		232	(◪) (CAPS OFF) H
i	i		233	(◪) (CAPS OFF) I
j	j		234	(◪) (CAPS OFF) J
k	k		235	(◪) (CAPS OFF) K
l	l		236	(◪) (CAPS OFF) L
m	m		237	(◪) (CAPS OFF) M
n	n		238	(◪) (CAPS OFF) N
o	o		239	(◪) (CAPS OFF) O
p	p		240	(◪) (CAPS OFF) P
q	q		241	(◪) (CAPS OFF) Q
r	r		242	(◪) (CAPS OFF) R

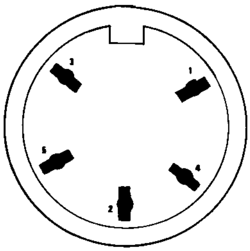
Atari ASCII Character Std.	Ext.	ASCII Character	Decimal Code	Keystrokes For Outputting Character
			243	(◪) (CAPS OFF) S
			244	(◪) (CAPS OFF) T
			245	(◪) (CAPS OFF) U
			246	(◪) (CAPS OFF) V
			247	(◪) (CAPS OFF) W
			248	(◪) (CAPS OFF) X
			249	(◪) (CAPS OFF) Y
			250	(◪) (CAPS OFF) Z
			251	(◪) CTRL-;
			252	(◪) SHIFT-=
			253	ESC/CONTROL-2
			254	ESC/CONTROL->
			255	ESC/CONTROL-BACKSPACE

Appendix C.
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	
CHR$		PADDLE	
CLOAD	CLOA.	PEEK	
CLOG		PLOT	PL.
CLOSE	CL.	POINT	P.
CLR		POKE	POK.
COLOR	C.	POP	
COM		POSITION	POS.
CONT	CON.	PRINT	PR. or ?
COS		PRINT#	PR# or ?#
CSAVE	CS.	PTRIG	
DATA	D.	PUT	PU.
DEG	DE.	RAD	
DIM	DI.	READ	REA.
DOS	DO.	REM	R. or .
DRAWTO	DR.	RESTORE	RES.
END		RETURN	RET.
ENTER	E.	RND	
EXP		RUN	RU.
FOR	F.	SAVE	S.
FRE		SETCOLOR	SE.
GET	GE.	SGN	
GOSUB	GOS.	SIN	
GOTO	G.	SOUND	SO.
GRAPHICS	GR.	SQR	
IF		STATUS	ST.
INPUT	I.	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.

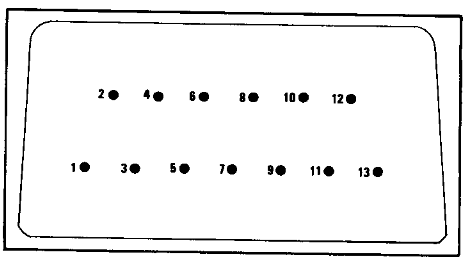
Appendix D. Pinouts

Monitor Jack



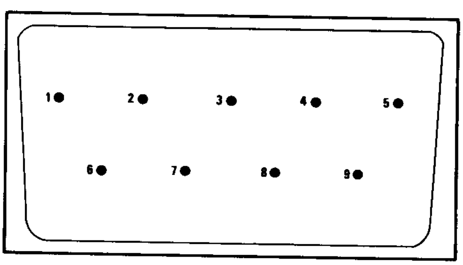
1. Composite Luminence
2. Ground
3. Audio Output
4. Composite Video
5. Composite Chroma

Serial I/O Jack



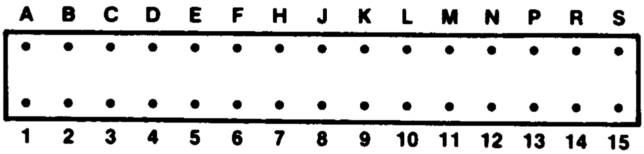
1. Clock Input
2. Clock Output
3. Data Input
4. Ground
5. Data Output
6. Ground
7. Command
8. Motor Control
9. Proceed
10. =5/Ready
11. Audio Input
12. =12 VOLTS
13. Interrupt

Joystick Jack



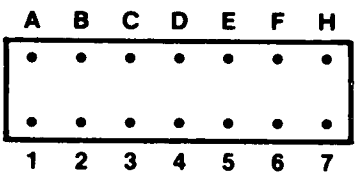
1. (Joystick) Forward Input
2. (Joystick) Back Input
3. (Joystick) Left Input
4. (Joystick) Right Input
5. B (Paddle) Input
6. Trigger Input
7. =5 VOLTS
8. Ground
9. A (Paddle) Input

Cartridge Slot



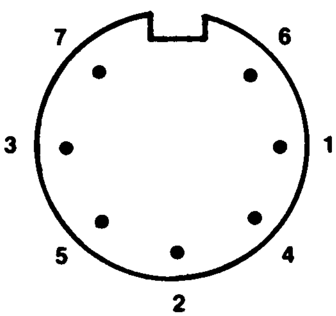
A RD4	1. S4
B GND	2. A3
C A4	3. A2
D A5	4. A1
E A6	5. A0
F A6	6. D4
H A8	7. D5
J A9	8. D2
K A12	9. D1
L D3	10. D0
M D7	11. D6
N A11	12. S5
P A10	13. +5V
R R1N	14. RD5
S B02	15. CCTL

Enhanced Cartridge Interface



A. Reserved	1. EXSEI
B. IRQ	2. RST
C. HALT	3. DIXX
D. A13	4. MPD
E. A14	5. Audio
F. A15	6. REF
H. GND	7. +5V

Power Adapter Plug



1. +5V
2. Shield
3. Ground
4. +5V
5. Ground
6. +5V
7. Ground

Appendix E.
Printer Usage with the Atari XE

A printer can be a valuable addition to an XE computer system, allowing it to perform a number of useful tasks. For example, a printer enables the XE to function as a word processor. Atari provides two ways to interface a printer to the system unit:

- Atari 800 interface modulue
- Peripheral expansion bus

Almost any parallel printer may be connected to the Atari XE after the interfae module has been correctly installed. Two of the more popular parallel printers used with the Atari are the Epson MX/RX-80 and the GEMINI.

As an alternative to the Atari 850 interface module, the consumer may elect to use the peripheral daisy chain. To date, only Atari markets printers that may be attached via the peripheral bus. These include the Atari 1020, Atari 1025, and Atari 1027.

A printer attached to the Atari in either of the preceding ways can be referenced using the device name "P:". The operating system only supports one printer device. Therefore, confusion will result if more than one active printer is attached to the peripheral chain.

The final way to connect a printer to an Atari system is to use one of the serial ports of the Atari 850 interface module. When so attached, the operating system will recognize the printer as "RX:". where X represents the number of the serial port (1-4).

LISTING PROGRAMS

The LIST command can output a copy of the program currently stored in the computer's memory. Since the printer is known as "P:", the LIST command requires this device name to cause the output to be sent to the printer.

```
LIST "P:"
```

OUTPUTTING DATA

A PRINT# statement is most commonly used to output data to the printer. However, an I/O filenumber must be opened for the printer before any data can be output. The following statement is a typical OPEN statement that can be used to establish an I/O filenumber for the printer:

```
OPEN #3, 8, 0, "P:"
```

The "8" designates the printer for output. "#3" is the filenumber. The following example program demonstrates the use of the OPEN and PRINT# statements to output data to the printer:

```
100 OPEN #1, 8, 0, "P:"
110 FOR J = 1 TO 15
120 PRINT# 1; J, J^2
130 NEXT J
140 CLOSE #1
150 END
```

If only intermittent printer output is necessary for a specific program, it is advisable to use LPRINT instead of PRINT#. The equivalent of the preceding program using LPRINT is given below:

```
100 FOR J = 1 TO 15
110 LPRINT J, J^2
120 NEXT J
130 END
```

Notice that no OPEN statement is required with LPRINT. Generally, PRINT# is used in place of LPRINT because PRINT# is faster than LPRINT. The speed difference is difficult to notice in a short program, but becomes apparent in more lengthy applications.

Appendix F.
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, PEEK (97) + 256 * 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 F.

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 F-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 = sideways 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 F-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 trigonometic functions are calculated using degrees or radians (0 = radians; 6 = degrees).

Table F-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 F-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

Appendix G. Memory Bank Switching

Table G.1. Bank Switching

Device States		Bank of "Extra" memory being accessed			
CPU	**Antic**	**#1**	**#2**	**#3**	**#4**
Extra	Extra	193	197	201	205
Extra	Normal	225	229	233	237
Normal	Extra	209	213	217	221
Normal	Normal	241	245	249	253

"Poking" the value indicated in table G.1 into location 54017 will have the described affect. For example,

```
POKE 54017,193
```

will set both devices to operate with extra memory bank #1.

Table G.2 shows the binary values corresponding with the decimal values in table G.1.

Table G.2. Decimal and Binary Equivalents

Decimal	Binary
193	11000001
197	11000101
201	11001001
205	11001101
225	11100001
229	11100101
233	11101001
237	11101111
209	11010001
213	11010101
217	11011001
221	11011101
241	11110001
245	11110101
249	11111001
253	11111101

Index

$15.95

ATARI® XE™ User's Handbook

Atari XE User's Handbook is a clear, concise, and practical guide to the Atari XE line of personal computers. This book covers introductory concepts designed to allow the "first time" computer user to operate and program the Atari XE in BASIC. This book also contains a great deal of information on advanced topics such as graphics, DOS usage, and file handling that are of interest to the experienced computer user.

The following topics are covered in depth in the **Atari XE User's Handbook**:

- Installation
- Operation
- Atari XE hardware
- Atari peripherals
- BASIC programming fundamentals
- File handling
- Graphics and sound
- DOS 2.0 and 2.5
- Printer usage
- Useful PEEKS and POKES
- Memory Bank Switching

Numerous examples and illustrations are provided throughout the book. **Atari XE User's Handbook** also includes a number of useful appendices and is fully indexed. No user of an Atari XE computer should be without the **Atari XE User's Handbook**.

ISBN: 0-938862-41-3 LC: 85-13896

WSi Weber Systems, Inc.
8437 Mayfield Road
Chesterland, OH 44026