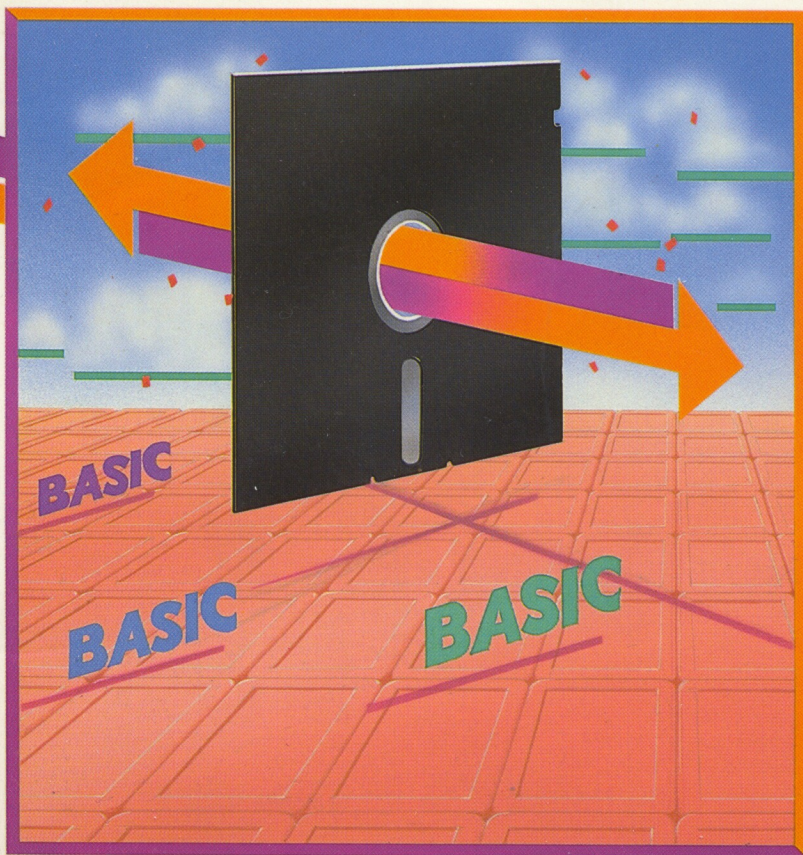


Computer Direct

312/382-5050

Programmer's Reference Guide for the ATARI® 400™/800™ Computers

David Heiserman



Programmer's Reference Guide for the ATARI[®] 400[™]/800[™] Computers

David Heiserman

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

Copyright © 1984 by David Heiserman

FIRST EDITION
FIRST PRINTING

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-22277-9
Library of Congress Catalog Card Number: 83-51616

Edited by *Welborn Associates*
Illustrated by *D.B. Clemons*

ATARI®, 400™ and 800™ are trademarks of and used by permission of Atari, Inc.

This book is published by *Howard W. Sams & Co., Inc.* which is not affiliated with Atari, Inc. and Atari is not responsible for any inaccuracies.

Printed in the United States of America.

**Programmer's Reference Guide
for the
ATARI® 400™/800™ Computers**



David L. Heiserman has been a freelance writer since 1968. He is the author of more than 100 magazine articles and 20 technical and scientific books. He studied applied mathematics at Ohio State University and is especially interested in the history and philosophy of science. He is the author of the Sams books *Intermediate Programming for the TRS-80* and *Intermediate-Level Apple II Handbook*.

Preface

The ATARI 400 and 800 Personal Computers are marvelous machines. The graphics system is, I think, incomparable; and it is no coincidence that the graphics in Chapters 4 and 5 occupy more space in this book than any other single topic. The I/O system is unusually flexible, and much of the material in this book deals with that part of the system; particularly Chapter 6.

The ATARI 400/800 Operating System holds its own in terms of quality and performance. And unlike some other brands of personal computers, the operating system is wide open for experimentation. Chapters 8 and 9 are especially meaningful in this regard, because they map the entire operating system and review the 6502 instruction set. These chapters indicate addresses and relevant data in both hexadecimal and decimal formats (the latter being more appropriate for use with ATARI BASIC).

This book is organized by general topics so that the reader can get access to certain information in the most effective way. That sort of format is especially useful for someone who is already familiar with the most common ATARI Home Computer features, and wants to know more about them. It is also a guide for users who are accustomed to other personal computers, but want to become familiar with the special features of the ATARI system in the shortest possible time (see especially Chapters 1, 2, and 3).

That is not to say that a beginner cannot learn anything from this book—quite the contrary. The organization of material in this book is most satisfactory for a beginner who has a desire to experiment with new ideas and techniques on a first-hand basis. Although such an individual might not fully understand the finer points of some topics, the mere effort guarantees a level of success that is often difficult to achieve from simpler, step-by-step presentations.

I would like to acknowledge the invaluable assistance offered by my personal secretary, Robin M. Yates. She prepared many of the tables and drawings, proofread the drafts, and assembled the final manuscript for us.

DAVID HEISERMAN

Contents

CHAPTER 1

Getting Started	13
SOME ATARI SYSTEM CONFIGURATIONS	15
A Minimum Working System	15
Minimum System Plus Program Recorder	16
Minimum System Plus One Disk Drive	17
A Common Multiple-Peripheral System	18
SOME SPECIAL KEYBOARD OPERATIONS	19
Some Control Keys	20
SOME SCREEN EDITING FEATURES	22
WORKING WITH THE PROGRAM RECORDER	23
Connecting the Program Recorder to the System	24
Saving and Loading Programs With the Program Recorder	24
ROUTINE DISK OPERATIONS	26
Displaying the Current Disk Directory	26
Returning to BASIC	28
Copying Files to the Same Disk	28
Deleting Files	30
Renaming Existing Files	31
Locking Files	32
Unlocking Files	32
Copying DOS Files	32
Formatting a Disk	33
Duplicating an Entire Disk	33
Getting MEM.SAV Onto a Disk	35
An Alternative File-Copying Operation	35
The Machine-Language Options	36
Saving and Loading Programs With the Disk Drives	36

CHAPTER 2

ATARI BASIC Notation, Rules, and Limitations	39
NUMERIC AND STRING CONSTANTS	39
Numeric Constants	40
String Constants	42
BASIC VARIABLES AND VARIABLE NAMES	44
Numeric Variables	45
String Variables	47
DIMENSIONING STRING VARIABLES AND NUMERIC ARRAYS	48
DIMensioning String Variables	49
Subscripted Numeric Variables and Arrays	51
OPERATIONS AND OPERATORS	55
Arithmetic Operators	55
Relational Operators	56
Logical Operators	62
Order of Precedence for Operators	64

CHAPTER 3

The BASIC Programming Language	67
A SUMMARY OF STATEMENTS, COMMANDS, AND FUNCTIONS	68
A SUMMARY OF ATARI BASIC SYNTAX AND APPLICATIONS ..	73

CHAPTER 4

The Text and Graphics Screens	115
FEATURES COMMON TO ALL SCREEN MODES	115
MODE-0 GRAPHICS	143
Organization of the Mode-0 Color Registers	144
Working With the Mode-0 Column/Row Format	145
Working With the Mode-0 Margins	147
Using the POSITION Statement	148
Alternative Column/Row Techniques	148
Using LOCATE, GET, and PUT in Mode 0	149
The Mode-0 Screen RAM Format	150
MODE-1 AND MODE-2 GRAPHICS	152
Organization of the Mode-1 and Mode-2 Color Registers	153
Accessing the Character Set From Modes 1 and 2	154
Working With the Mode-1 and -2 Column/Row Format ...	155
Using the POSITION Statement	159
Alternate Column/Row Techniques	160
Using LOCATE, GET, and PUT in Modes 1 and 2	165
The Mode-1 and Mode-2 Screen RAM Formats	166
Full-Screen Formats	168
CUSTOM CHARACTERS SETS FOR MODES 0, 1, AND 2	179

THE FOUR-COLOR MODES: 3, 5, AND 7	182
Working With the Color Registers	183
Column/Row Screen Formats and Graphics Operations ...	184
The Screen RAM Address Formats and Operations	188
TWO-COLOR MODES 4 and 6	199
Working With the Color Registers for	
Screen Modes 4 and 6	199
Column/Row Screen Formats and Operations for	
Modes 4 and 6	204
Screen RAM Formats and Operations for Modes 4 and 6 ..	209
THE 2-COLOR MODE-8 SCREEN	216

CHAPTER 5

Player/Missile Graphics	239
PLAYER AND MISSILE CONFIGURATIONS	240
Bit Maps for the Player Figures	241
Bit Maps for the Missile Figures	243
The Overall Player/Missile Bit Map	245
Setting the Starting Address of the Player/Missile	
Bit Map	247
Protecting the Player/Missile Bit Map	249
ADJUSTING THE WIDTH OF THE PLAYER/MISSILE FIGURES ...	250
SETTING PLAYER/MISSILE COLORS	251
INITIATING AND TERMINATING PLAYER/MISSILE	
GRAPHICS	251
MOVING THE PLAYER/MISSILE FIGURES	254
PLAYER/PLAYFIELD PRIORITIES	255
COLLISION DETECTION	257

CHAPTER 6

More About I/O Operations	263
WORKING WITH THE PROGRAM RECORDER I/O	263
ATASCII-Coded BASIC Program—LIST "C:"	
and ENTER "C:"	264
Nonprogram Files—PRINT and INPUT	266
Nonprogram Files—PUT and GET	268
WORKING WITH THE DISK DRIVE I/O	270
ATASCII-Coded BASIC Programs—LIST "D:"	
and ENTER "D:"	270
Nonprogram Files—PRINT and INPUT	272
Nonprogram Files—PUT and GET	274
SAVING, LOADING, AND RUNNING	
BINARY FILES UNDER DOS	275
Saving Binary Programs and Data	276
Loading and Running Binary Programs	277
OPENING AND CLOSING IOCB CHANNELS	278

USING THE XIO COMMAND	283
Controlling Outgoing Lines With XIO 34	285
Configuring Baud Rate, Word Size, and Stop Bits with XIO 36	285
Setting Translation Modes and Parity with XIO 38	286

CHAPTER 7

A Miscellany of Principles and Procedures	289
MORE ABOUT THE SOUND FEATURES	289
The SOUND Statement	290
Reproducing Musical Scores	292
Experimenting With Sound Effects	292
MORE ABOUT THE USR FUNCTION	293
Passing Values to a Machine Routine	296
Passing Values From a Machine Language Routine	300
SCREEN DISPLAY LISTS	300
The ANTIC Instruction Set	300
Structure of a Display List	303
Locating the Display List	306
A LOOK AT TOKENIZED BASIC	307

CHAPTER 8

The ATARI Memory Map	311
ZERO-PAGE AND STACK RAM	313
OPERATING SYSTEM AND BASIC RAM	322
Operating System RAM: 512-1151	322
BASIC System RAM: 1152-1535	337
DOS RAM USAGE: 1792-10879	338
BASIC ROM AREA: 40960-49151	338
HARDWARE I/O ROM AREA: 53248-55295	340
The CTIA (or GTIA) Device	341
The POKEY Device	343
The PIA Device	345
The ANTIC Device	345
OPERATING SYSTEM ROM AREA: 55296-65535	346

CHAPTER 9

The 6502 Instruction Set	349
---------------------------------------	------------

APPENDIX A

Number-System Base Conversions	381
HEXADECIMAL-TO-DECIMAL CONVERSIONS	382
DECIMAL-TO-HEXADECIMAL CONVERSIONS	384
CONVENTIONAL DECIMAL TO 2-BYTE DECIMAL FORMAT ...	385

TWO-BYTE DECIMAL TO CONVENTIONAL	
DECIMAL FORMAT	387
BINARY-TO-DECIMAL CONVERSION	387
BINARY-TO-HEXADECIMAL CONVERSION	388
HEXADECIMAL-TO-BINARY CONVERSION	390
DECIMAL-TO-BINARY CONVERSION	390
A COMPLETE CONVERSION TABLE FOR DECIMAL 0-255	391

APPENDIX B

ATARI BASIC Reserved Words and Tokens	395
--	------------

APPENDIX C

ATARI Character Codes	401
------------------------------------	------------

APPENDIX D

ATARI Keyboard Codes	425
-----------------------------------	------------

APPENDIX E

Screen RAM Addressing Ranges for the	
 ATARI Screen Modes	435

APPENDIX F

Derived Trigonometric Functions	469
--	------------

APPENDIX G

ATARI Error and Status Codes	471
---	------------

APPENDIX H

ATARI 400/800 Hardware Details	479
---	------------

Index	489
--------------------	------------

Chapter 1

Getting Started

The ATARI® 400™/800™ personal computer systems are adaptable to a wide range of configurations and possible operating schemes. The purpose of this opening chapter is to introduce some of those features.

Figs. 1-1 and 1-2 show the basic console units for models 400 and 800, respectively. One obvious difference between them is the nature of the keyboard. The ATARI 400 Home Computer uses a film, touch-sensitive keyboard, while the 800 model uses an ordinary mechanical-key mechanism.

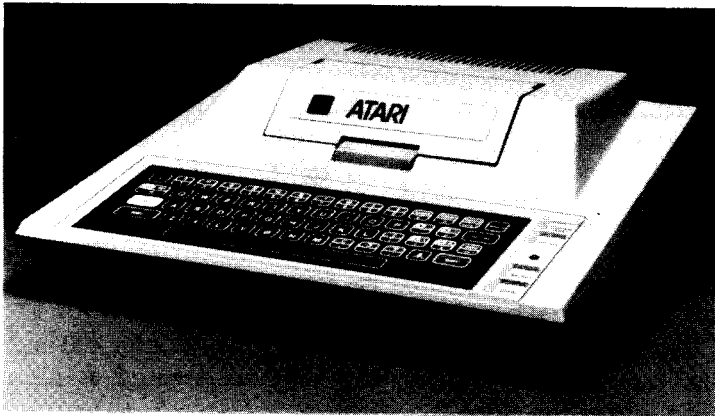


Fig. 1-1. The ATARI 400 Home Computer console. (Courtesy Atari, Inc.)

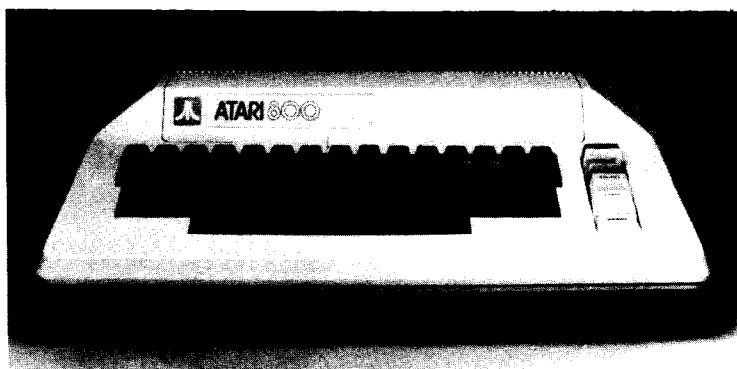


Fig. 1-2. The ATARI 800 Home Computer console. (Courtesy Atari, Inc.)

There are a couple of less-obvious differences. For one, the ATARI 800 Home Computer can accept two different program cartridges, while the 400 can accept only one.

Fig. 1-3 shows the arrangement of jacks and switches along the right-hand side of the ATARI 800 console unit. The arrangement for the model 400 is similar, but does not include the MONITOR jack. That simply means that the 400 must use an ordinary tv receiver as its display screen, while the 800 offers the option of using a tv receiver or a monitor.

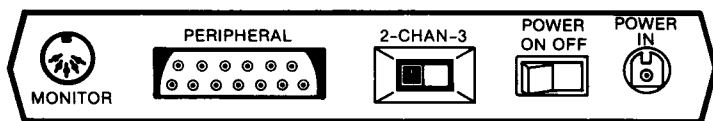


Fig. 1-3. Connection panel for the Model 800. The Model 400 is identical except for the lack of a MONITOR jack.

For the most practical purposes, however, the 400 and 800 models are identical, and they are treated as such throughout this book.

Also, it will be assumed through this book that the ATARI BASIC cartridge is installed. As far as the 400 model is concerned, that means plugging it into the only slot that

is available for that purpose—under the hatch just above the keyboard assembly. For the 800 unit, the BASIC cartridge must be installed in the left-hand cartridge slot. (The discussions further assume that the 800's right-hand cartridge slot is unused.)

SOME ATARI SYSTEM CONFIGURATIONS

The individual components of an ATARI system must be connected in certain ways if that overall system is to function properly. This section outlines several commonly used systems.

A Minimum Working System

Fig. 1-4 illustrates the minimum working ATARI system: a console unit and an ordinary tv receiver.

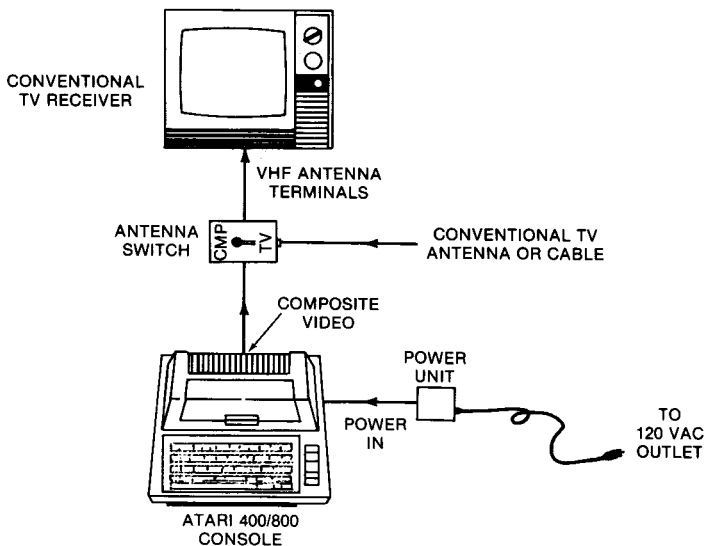


Fig. 1-4. Diagram of the simplest ATARI Home Computer system.

The power-transformer assembly is plugged into an ordinary 120 V ac outlet, and the smaller plug goes into the POWER IN jack on the side of the console unit.

A cable and plug attached to the rear of the console goes to the antenna terminals on a standard tv receiver. If you plan to use the tv for viewing ordinary programming, that antenna connection should be made through an antenna switch that is provided with the basic unit.

To get this configuration into operation:

1. Turn on the tv set.
2. If you are using the antenna switch, set the switch to its COMPUTER position.
3. Turn on the computer console, using the POWER switch located on the left-hand side of the unit.
4. Set the CHAN switch on the side of the console for either Channel 2 or 3, and match the tv channel selector accordingly. Use the channel that can be tuned for the lesser amount of outside interference.

When all is going well, the tv will show a blue background color, a lighter blue READY message, and a light-blue square just below the message. At that time, the system is ready to operate in ATARI BASIC.

If you have an ATARI 800 system and wish to work with a monitor unit instead of a tv receiver, the hardware arrangement is somewhat simpler. You have no need for the antenna switch nor the tv cable coming from the back of the console. Simply run the monitor cable from the 5-slot MONITOR jack on the side of the console to the video and audio input terminals on the monitor. The CHAN selector switch setting is not relevant at all.

Minimum System Plus Program Recorder

Fig. 1-5 shows the minimum system configuration as extended to include the ATARI 410 program recorder. Using the program recorder enables you to save programs on the magnetic tape in a standard tape cassette, and then reload those programs at some later time.

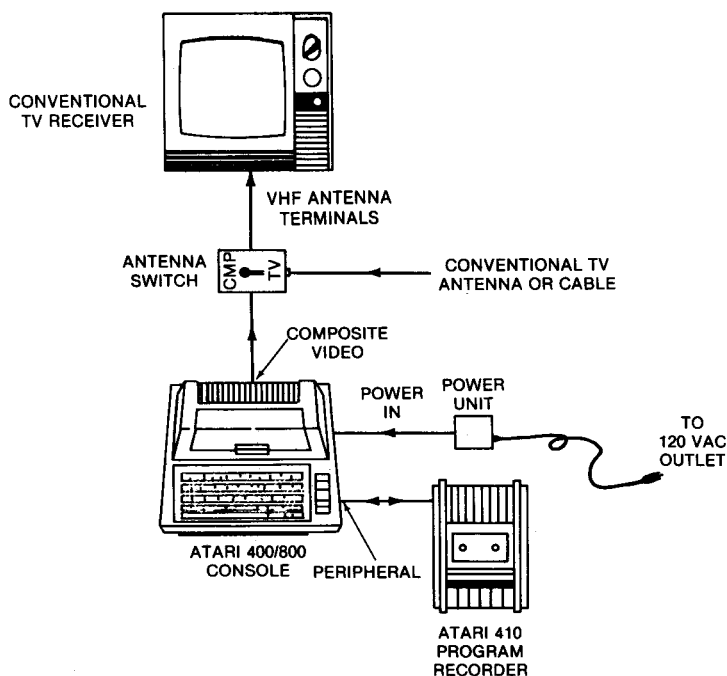


Fig. 1-5. Connection diagram for an ATARI Home Computer system that is using the 410 program recorder.

Connect the program recorder to the PERIPHERAL jack on the side of the console unit, using a cable assembly that is provided for that purpose.

The start-up procedures are identical to those already described for the minimum system.

Minimum System Plus One Disk Drive

A single ATARI 810 disk-drive unit is connected to the console in the same way that the program recorder is—directly to the PERIPHERAL jack. The start-up procedure includes two additional steps, however.

First, when installing a disk-drive, make certain that the slide switches on the back of the drive unit are set for drive No. 1.

Turn on the tv receiver or monitor, and then insert a diskette into the disk-drive unit. That diskette must be properly formatted and include the DOS system programming (the DOS system disk supplied with the disk assembly will include that programming).

Next, turn on the disk-drive unit. Do not have the console unit turned on at this time. Only when the disk drive stops running (the red light will go out) should you turn on the console.

A Common Multiple-Peripheral System

Fig. 1-6 illustrates one of the most common ATARI configurations: the basic console unit, a tv receiver or monitor, a single disk-drive unit, a model 850 serial interface module, and a printer.

The start-up sequence for such arrangements is quite critical. In this particular case:

1. Turn on the tv set or monitor.
2. Insert a diskette containing the DOS system programming into the disk-drive unit.
3. Turn on the disk-drive unit and wait for the drive to stop running.
4. Turn on the 850 serial interface module.
5. Turn on the printer.

There are many other possible configurations that can be far more complex than these—using multiple disk drives, for instance. Consult the manuals that are supplied with the peripherals for exact details.

Generally speaking, however, the turn-on procedure follows the same general plan: turn on the tv or monitor, boot DOS by turning on disk-drive 1, turn on the console, turn on the interface module, and turn on peripherals that are connected through the interface module.

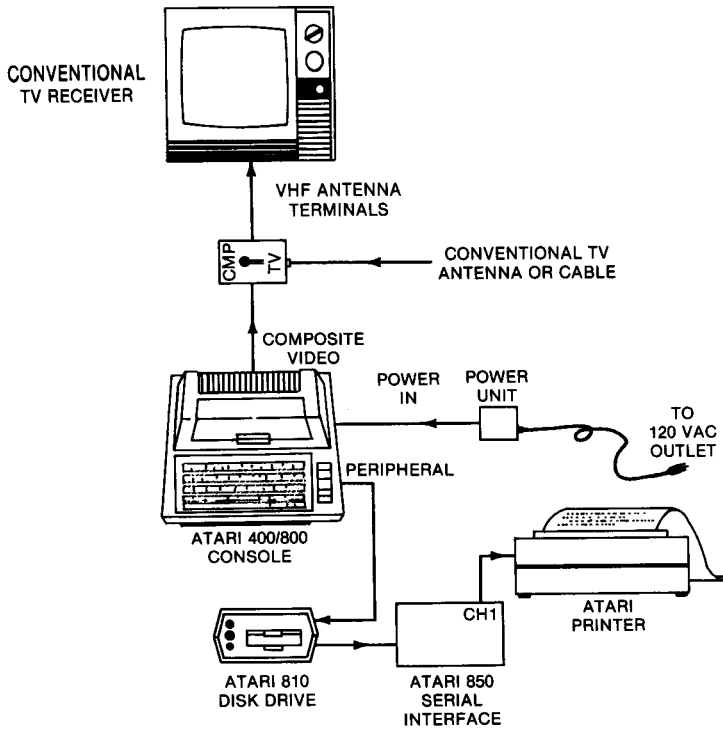


Fig. 1-6. Connection diagram for an Atari Home Computer system that uses a single 810 disk drive, an 850 serial interface module, and an Atari Home Computer printer.

SOME SPECIAL KEYBOARD OPERATIONS

When an Atari system is properly configured and initialized, you should see a blue screen, BASIC's READY prompting message, and the cursor (the light blue square under READY). That means that the system is ready for commands from the keyboard.

Much of the material remaining in this book deals with the nature of those BASIC commands. During the course of working with those commands, however, it is often necessary or desirable to execute some special keyboard functions.

Some Control Keys

Most of the ATARI computer keyboard is identical to that of a conventional typewriter. There are some special keys and key functions that are generally irrelevant for ordinary typing operations, but quite important for operating a computer.

The **RETURN** key is perhaps the most-used control key. You must strike the control key whenever you want the computer to execute a command that you've given it in a typewritten form. When you are ready to execute a BASIC program, for example, you should type RUN on the keyboard, and then strike the **RETURN** key to get the computer to read and execute the command. The special control-key operations described in this section do not have to be followed by a **RETURN** keystroke, however.

The two **SHIFT** keys serve much the same function as the shift keys on an ordinary typewriter. There are some differences, though. For instance, when you turn on the ATARI system, you find that all letters of the alphabet are printed to the screen in an upper-case format. Depressing a **SHIFT** key while typing letters of the alphabet will cause the system to print lower-case letters. That is just the reverse of ordinary typewriters.

It is possible to change the format by striking the **CAPS LOWR** key. Having done that, all letters of the alphabet are normally printed in their lower-case form; and you must hold down one of the **SHIFT** keys in order to print upper-case letters.

To return to normal upper-case printing, hold down one of the **SHIFT** keys and strike the **CAPS LOWR** key again.

The **CAPS/LOWR** key affects only the letters of the alphabet. That is another feature that makes the computer different from a typewriter. No matter what the **CAPS/LOWR** setting might be, striking the **3** key will print a 3, and holding down a **SHIFT** key while striking the **3** key will print a “ to the screen.

As mentioned earlier, the screen shows light-blue characters against a darker blue background. You can reverse the situation by striking the ATARI logo key—the one marked with the familiar ATARI symbol. Strike that key again, and the colors return to normal.

The **CTRL** key, in effect, multiplies the number of key functions that are available. Holding down the **CTRL** key while striking another key often changes its function. Throughout this book, a statement such as **CTRL-A** means: hold down the **CTRL** key while striking the **A** key.

The **DELETE BACK S** key is intended for programmers who make typing errors. Hold down a **SHIFT** key while striking the **DELETE BACK S** key, and the computer deletes the entire line of text that is marked by the cursor. Strike the **DELETE BACK S** key without holding down a **SHIFT** key, and the computer erases the character under the cursor and moves it one column, or character location, to the left.

The **INSERT** key will create a blank line for inserting a new line of printed text—if you are holding down a **SHIFT** key at the time. Make that keystroke without holding down one of the **SHIFT** keys, and the computer will print the > symbol on the screen. To insert a single character into a line of text, do a **CTRL-INSERT** operation.

The same general idea applies to the **CLEAR** key. Hold down a **SHIFT** key while striking the **CLEAR** key, and the computer will clear the screen and set the cursor to its *home* position in the upper left-hand corner of the screen.

The four arrow keys located near the right side of the keyboard allow you to move the cursor to any desired point on the screen. In order to use them, however, you must be holding down the **CTRL** key at the same time.

The **CLR SET TAB** key serves the purpose of the tab functions on an ordinary typewriter. Strike that key alone, and the cursor will jump to its next horizontal tab location on the screen. You can set new tab positions by first setting the cursor to the desired tab position, and then striking the **CLR SET TAB** key while holding down one of the **SHIFT** keys. Finally, you can clear a current tab setting by holding down both the **CTRL** key and one of the **SHIFT** keys while striking the **CLR SET TAB** key.

The **BREAK** key is used for stopping the execution of a program.

The **ESC** key, like the **CTRL** key, changes the normal functions of other keys. Whereas you must hold down the **CTRL** key while striking another key, you use the **ESC** key in sequence: first strike the **ESC** key, release it, and then strike another key. The special **CTRL** and **ESC** operations are described in later discussions in this book.

SOME SCREEN EDITING FEATURES

When working in BASIC, it is often necessary to change some of the material printed in the program. The ATARI system offers some program-editing features that make the task much simpler.

The most important point to bear in mind is that the ATARI screen is "live." That is to say, what you see on the screen is what is actually in program memory. So change something on the screen, and you also change the programming as well.

This leads to the notion of editing a program by first listing the relevant portions of it on the screen, using the **CTRL**-arrow keys to position the cursor, and then using the insert/delete operations to change the text. Once you have changed a line of programming, you strike the **RETURN** key in order to enter that change into memory.

WORKING WITH THE PROGRAM RECORDER

The ATARI 410 program recorder offers the most economical means for saving and loading programs and other kinds of data. When saving programs and data by means of the program recorder, the ATARI system converts the information into audio tones that can be easily recorded on magnetic tape; specifically on the narrow tape in ordinary audio cassettes. And when loading previously saved information, the program recorder reproduces the audio tones, and the system converts them back into meaningful computer data for the system's RAM.

It is possible, and certainly economically attractive, to save more than one program on a single cassette. The only problem is being able to find the segment of tape that contains the desired program. For that reason, the ATARI program recorder includes a numerical tape-counter mechanism.

Whenever you are starting to work with a cassette, it is a good idea to rewind it to the beginning and reset the tape counter to zero. Then when you are ready to record a program from the ATARI computer memory, note the tape-counter reading before starting the recording operation. Write down that reading as well as a short, but meaningful, description of the program. Then note the tape-counter reading at the end of the recording session so that you will know where to begin recording another program on the same tape at some later time.

If you wish, you can listen to the audio tones during a data recording or playback session. If you are using an ordinary tv set as a screen monitor for the system, simply turn up the volume control; or if you are using a monitor, make sure that the audio plug is inserted into the monitor's audio input jack, and turn up the volume control on that unit.

Connecting the Program Recorder to the System

Fig. 1-5 shows the arrangement of the system if you are using no peripheral devices other than a tv or monitor and the program recorder. The program recorder plugs directly into the peripheral connector on the side of the ATARI console unit.

If the program recorder is used in conjunction with several other peripheral devices, it must be connected to the system through a serial interface module, and it must be the last peripheral in line.

Saving and Loading Programs With the Program Recorder

The most commonly used command for saving BASIC programs on cassette is CSAVE; and getting a BASIC program from tape and into the ATARI computer is by means of the CLOAD command.

To save a BASIC program on the program recorder, first use the FAST FORWARD and REVERSE keys on the program recorder to find the end of any programming that currently exists on the tape. Note the reading on the recorder's tape counter for future reference.

Set the recorder to its record mode by depressing both the RECORD and PLAY levers; and immediately follow that by entering the following command at the ATARI console:

CSAVE

The recording session will require at least 30 seconds. Turn up the volume on the tv set or monitor if you want to listen to the steady whistle of the leader and the obnoxious sound of the data being transferred. Whether you wish to listen to the sounds or not, you know that the recording session is done when the BASIC prompt symbol and READY message reappear on the screen. Turn off the program recorder at that time.

You can load a BASIC program from cassette tape to the ATARI's program memory with the help of the CLOAD command. You must, however, first cue the tape to the beginning of the program you want to load.

Assuming that you have saved the reading from the tape counter as suggested for the CSAVE routine, find that location on the tape by entering the CLOAD command at the console and, upon hearing two beeps from the console loudspeaker, cue the tape to the beginning of your program.

Next, depress the PLAY lever on the program recorder and strike any key on the ATARI console (except the **BREAK** key)—that will begin the loading operation. You can listen to the audio activity through the loudspeaker of the tv or monitor unit; and if you do, you should hear a rather long, steady leader tone, followed by a lot of whistles and beeps that represent the data. The loading session is done when the sounds end or, if you choose not to listen to them, when the BASIC cursor and READY message reappear on the screen.

As is the case with most program-recorder operations, the system automatically selects IOCB Channel 7 for this one. That channel must not be open for any other purpose at the time you execute a CSAVE or CLOAD command. If you run into difficulties in this regard, try executing an LPRINT or CLOSE command before doing the CSAVE or CLOAD.

See Chapter 6 for some alternative techniques and commands for saving and loading all kinds of data.

ROUTINE DISK OPERATIONS

Fig. 1-6 shows a disk-drive unit connected to the ATARI system. When using more than one disk drive, determine the proper installation procedures from the user's manual. Assuming that DOS is properly booted as described earlier in this chapter, typing DOS and striking the **RETURN** key brings up the DOS menu—it is a convenient guide to running variety disk utility operations. See the two common versions of the DOS menu in Chart 1-1.

Chart 1-1. Two Common Versions of the DOS Menu

DOS Version 1.0			
A.	DISK DIRECTORY	I.	FORMAT DISK
B.	RUN CARTRIDGE	J.	DUPLICATE DISK
C.	COPY FILE	K.	BINARY SAVE
D.	DELETE FILE(S)	L.	BINARY LOAD
E.	RENAME FILE	M.	RUN AT ADDRESS
F.	LOCK FILE	N.	DEFINE DEVICE
G.	UNLOCK FILE	O.	DUPLICATE FILE
H.	WRITE DOS FILE		
DOS Version 2.0S			
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.SAVE
G.	UNLOCK FILE	O.	DUPLICATE FILE
H.	WRITE DOS FILE		

Displaying the Current Disk Directory

Execute the DOS command from the keyboard, and then select menu option A. The system will respond by printing this prompting message:

DIRECTORY—SEARCH SPEC, LIST FILE?

From there, you can see the complete listing of files on the current disk by striking the **RETURN** key twice in succession or entering this response:

D1:

If you have more than one disk drive, you can see the directory for drive number 2 by responding to the prompting message by entering this:

D2:

And if you have an ATARI printer connected to the system and want to print a disk directory to it, modify your response to this form:

D1:;P

The directory includes the filenames and the number of disk sectors each occupies. The directory concludes with a message that indicates the number of unused sectors.

A single diskette can hold as many as 64 files in its directory, and that many filenames cannot fit onto the display screen. It is possible to search and display the name of a single file by modifying your response to the prompting message to include the desired filename.

Suppose, for example, you want to know whether or not a filename SILLY.FUN is on the current disk. When you see the prompting message, respond by entering a:

D:SILLY.FUN

If, indeed, that file is present, the system will print the name again, along with the number of sectors devoted to it. If the file is not on the current disk, the system will simply print the number of free sectors remaining on the disk.

It is also possible to list just certain kinds of file names. Perhaps you want to list only those files that end with the extension .BAS. In that case, respond to the prompting message with this sort of entry:

D:*.BAS

In a manner of speaking, the asterisk tells the system to work with any combination of letters and numerals in that part of the file name.

By way of another example, suppose that you want to see the directory of all file names that begin with the letters ALT. This sort of response will do that for you:

D:ALT*.*

Returning to BASIC

The all-around safest way to return to BASIC from the DOS menu is to strike the **SYSTEM RESET** key. Alternatively, you can elect menu item B, but there is a chance that you will lose some BASIC programming that is resident in the system.

Actually the purpose of DOS menu item B is to return program control to the resident cartridge. If that happens to be the BASIC cartridge, the system begins running in BASIC. Or if it is a special language or game cartridge, selecting menu item B begins execution of that program.

Copying Files to the Same Disk

DOS menu item C provides a means for copying files and programs onto their own disk. This feature is most often used for generating backup copies on the same diskette. Perhaps you are planning to revise a program that already exists on the disk. You want to save the original version, but create a copy that you can modify. That is the primary application of DOS menu selection C.

Upon entering that selection, you will see this prompting message:

COPY FILE—FROM,TO?

The system is expecting you to enter two filenames separated by a comma. The first name is one that is to be copied and already in the disk directory. The second name is the name of its copy. The two names must be different; perhaps different only in the extension, but nevertheless different (see the general rules for composing disk file names described later in this chapter).

So if you want to copy a file that already has the name **BIGTIME** and give that copy the name of **BIGTIME.BAK**, this is the sort of response you should enter:

BIGTIME,BIGTIME.BAK

If you are using DOS version 2.0S and you have a BASIC program resident in the ATARI system, you will see the following kind of prompting message before the actual copying operation takes place:

TYPE "Y" IF OK TO USE PROGRAM AREA
CAUTION: A "Y" INVALIDATES MEM.SAV

The idea here is that the copying procedure might destroy any BASIC programming that is resident in the system at the time. If you are willing to lose that programming, then a "Y" response is appropriate. Otherwise, it is better to respond with an "N" and save the resident program on disk before attempting the copying routine again.

If you are using more than one disk drive, menu option C allows you to copy files from one disk to the other. In this instance, you can use the same file name if you choose, just making sure to indicate the source and destination drives in your response. For example:

COPY FILE—FROM,TO?
D1:BIGTIME,D2:BIGTIME

That will copy a file named **BIGTIME** from drive 1 to drive 2.

You can also use global (or “wild-card”) file names under the file-copying option. Suppose that you want to transfer all of the SYS, or system, routines from drive 2 to drive 1. The appropriate response to the prompting message is:

D2:*.SYS,D1:*.SYS

Or if you want to copy all files beginning with FUS to the same disk, but add a BAK extension, the appropriate entry is:

FUS*.*,FUS*.BAK

Finally, there is the option of merging disk files—adding the content of a source file to the end of an existing destination file. The only catch is that both files must have been saved in an ATASCII format (see Chapter 6). The general syntax of such an operation is:

source,dest /A

where *source* is the file name for the source file, and *dest* is the file name of the destination file. The suffix /A is what prevents the source file from completely writing over the destination file.

Deleting Files

There is little point in cluttering valuable disk space with files and programs that are no longer of any use. DOS menu option D allows you to delete, or erase, any unwanted programs or files.

Upon selecting menu option D, the system displays this prompting message:

DELETE FILE SPEC

You should respond by entering the name of the file to be deleted. And if you enter a file name that exists on the disk, the system will ask you whether or not you really want to delete it:

TYPE “Y” TO DELETE . . .

If you change your mind about deleting the file, simply respond by entering an N. Respond with a Y, and the system will delete the file you named.

Use the asterisk option to cite global file names. The extreme case is that of deleting all the files on a disk. To do that, respond to the original prompting message by entering:

.

and enter a Y in response to the “are-you-sure” prompting message. Herein lies a certain problem: the system brings up the “are-you-sure” message for every program or file that it is about to delete. It can be quite troublesome to respond with a Y for each one to be deleted. The way around that inconvenience is to append the *.* response with /N. For example:

. /N

The /N overrides the “are-you-sure” prompting feature.

Renaming Existing Files

DOS menu option E gives you an opportunity to rename an existing file. Elect that option, and you will see this sort of prompting message on the screen:

RENAME—GIVE OLD NAME,NEW

The general idea is to respond by citing the old file name followed by the revised name. If the old name does not exist, or if that file is locked, the system returns an error code. But you are in trouble if you inadvertently cite a NEW name that is identical to one that already exists on the disk. You end up with two different files having the same name, and the system has no way of distinguishing them.

Locking Files

DOS menu option F allows you to lock a file so that it cannot be changed or renamed as long as the file remains locked. Responding with that menu selection brings up this prompting message:

WHAT FILE TO LOCK?

Simply respond by entering the name of the file—one that already exists in the directory.

You can use the global option to lock all files having a specified name or extension in common. It is a good idea, for example, to lock all SYS files so that they cannot be inadvertently changed, erased, or renamed. Do that by responding with:

*.SYS

Unlocking Files

There are instances where it is necessary to unlock a previously locked file; perhaps to modify it or erase it altogether. DOS menu option G offers this feature. Selecting that option brings up this sort of prompting message:

WHAT FILE TO UNLOCK?

Simply respond with the name of the file to be unlocked, and the computer will take care of the task for you.

Copying DOS files

A finished disk of BASIC programs ought to contain the system files that boot DOS for you. That circumvents the troublesome need to boot DOS with a special systems disk and then replacing it with your working BASIC disk. Electing DOS menu item H does that for you. The mechanical procedures are different for DOS 1.0 and 2.0S.

If you are using version 1.0, the system prints this prompting message:

TYPE "Y" TO WRITE NEW DOS FILE

Insert a properly formatted disk into drive number 1, and respond by entering a Y. That will copy the DOS.SYS program to the disk.

If you are running under DOS version 2.0S, the system prints this prompting message:

DRIVE TO WRITE FILES TO?

Insert a properly formatted disk into one of the disk-drive units, and respond with the drive number. The system brings up a TYPE "Y" TO WRITE prompting message; and if you respond with Y, it copies DOS.SYS and DUP.SYS to the designated disk.

Formatting a Disk

DOS menu item I is the one that formats a new disk for you. Every disk must be formatted before it can be of any practical use. Upon entering that option, the system responds with this sort of prompting message:

WHICH DRIVE TO FORMAT

Insert the disk to be formatted into one of the disk-drive units, and respond to the prompting message with the disk-drive number. The system will then display a TYPE "Y" TO FORMAT message to confirm that everything is set up properly. If that is the case, enter a Y, and the system will format the disk for you.

Duplicating an Entire Disk

DOS menu selection C allows you to copy disk files and programs one at a time. Menu selection J copies an entire disk. Selecting DOS menu item C brings up this prompting message:

DUP DISK—SOURCE,DEST DRIVES?

The general idea is to respond with the disk-drive number of the disk to be copied (the source disk), a comma and the disk-drive number of the new copy (the destination disk). The appropriate response and operations from that point are slightly different, depending on whether you have one or two disk drives available.

If you have just one disk drive available, you must respond to the prompting message by entering 1,1. That, in effect, says that the source and destination disks will both be in drive 1. Obviously you cannot fit two disks into the same drive unit, so the system will respond with this message:

INSERT SOURCE DISK, TYPE RETURN

Insert the disk to be copied and strike the **RETURN** key. The system will load up the ATARI RAM with disk data; and when it is full, you will see this message:

INSERT DESTINATION DISK, TYPE RETURN

Respond by replacing the source disk with a clean and formatted disk, and striking the **RETURN** key. The system will then load the most recent blocks of data to that disk.

When using a single disk drive in this fashion, it is often necessary to alternate the source and destination disks several times. This swapping operation will come to an end only after the entire contents of the source disk is copied to the destination disk.

Things are a lot simpler for the operator when two disk drives are available. Electing the disk copy operation, as before, brings up this prompting message:

DUP DISK—SOURCE,DEST DRIVES?

Respond by entering two different disk-drive numbers: the source followed by the destination. Suppose that you want to copy the content of the disk in drive 1 to a fresh disk in drive 2. That being the case, the appropriate response to that prompting message is:

1,2

After making that response, the system will print:

INSERT BOTH DISKETTES, TYPE RETURN

The computer then takes over the copying task completely. Usually you will find the disk drives running alternately through several cycles.

Getting MEM.SAV Onto a Disk

When you are working under DOS version 2.0S, electing menu item N will cause the system to generate the MEM.SAV file onto the disk in drive number 1. (Although the DOS version 1.0 lists an operation for menu item N, it is not functional.)

Most of the DOS menu operations use generous portions of the RAM area. That is not a problem if you have no vital programming in RAM at the time. But if you want to preserve a resident BASIC program you either have to save that program on disk before beginning DOS menu operations (the only option available under DOS version 1.0) or including the MEM.SAV routine on the current disk.

If MEM.SAV is on the current disk, the system will automatically use it to save portions of RAM-based programming that might otherwise be lost. Then when you leave the DOS menu operations, MEM.SAVE returns the original programming to RAM.

An Alternative File-Copying Operation

Menu item O is most useful when it is necessary to copy a file from one disk to another when you have just one disk drive unit. Menu item C lets you copy from one disk drive to another, or on the same disk if the file names are somehow different; but that selection cannot copy the same file name on two different disks that are alternately inserted into the same drive unit.

Upon selecting DOS menu item O, the system prints this prompting message:

NAME OF FILE TO MOVE?

Respond with the file name of the file to be copied. Having done that, the system will prompt you to insert the source disk and strike the **RETURN** key. After that, it prompts you to replace the source disk with the destination disk—and strike the **RETURN** key. If the file is a fairly long one, you will have to switch the source and destination disks several times.

The Machine-Language Options

DOS menu items K, L, and M deal with machine-language program operations. Chapter 6 describes these operations in detail.

Saving and Loading Programs With the Disk Drives

Starting with a properly formatted diskette, simply insert the diskette into a drive unit and enter a command of this general form:

SAVE "D[n]":*filename*[.ext]

That command includes a couple of optional expressions that are included in brackets. Expression *n* is necessary only when (1) using more than one disk drive and (2) you want to save the program on a drive other than drive No. 1. The *filename* is not optional, and it must follow these general rules:

1. It must be composed of nothing but numerals and capital letters.
2. It must be no more than eight characters long.
3. It must begin with a letter of the alphabet.

The file name extension *ext* is optional. It can be composed of no more than three characters, but they can be all numerals, all upper-case letters, or any combination of the two. The extension is separated from the file name with a period.

Normally, the extensions are used for specifying the type of file being saved. The following list suggests some commonly used file extensions and their meaning.

- .ASM—an assembly language source file
- .BAS—a BASIC program
- .BAK—a backup file
- .DAT—a DATA file
- .LST—an ATASCII-coded program file
- .OBJ—an object-code file
- .SCH or TMP—a “scratch,” or temporary file
- .TXT—a text file

Avoid using the SYS file extension, however, because the ATARI operating system uses that one for much of its own internal programming.

Executing the SAVE command will cause the disk drive to run and the upper red lamp to turn on. The routine is complete when the disk drive stops running and the uppermost lamp goes off.

You can load a tokenized BASIC program from the disk system by using a LOAD command of this form:

LOAD“D[*n*]:*filename*[*ext*]”

where *n* is the optional disk-drive number, *filename* is the name of the file to be loaded, and *ext* is the optional extension.

Assuming that the designated file exists on the current diskette, executing the LOAD command will cause the disk system to run until the program is completely loaded.

LOAD used IOCB Channel 7. If Channel 7 is already open for some other purpose when you execute a LOAD “D: command, DOS will automatically close and reopen Channel 7 for disk I/O operations.

Chapter 2

ATARI BASIC Notation, Rules, and Limitations

Most people begin familiarizing themselves with the ATARI 400/800 systems through the BASIC programming language, so this chapter introduces some of the fundamental concepts of that language. Although it is assumed that you are running the system with the ATARI BASIC COMPUTING LANGUAGE cartridge (CXL4002) connected into the single cartridge slot (Model 400) or the left-cartridge slot (Model 800), many of the principles apply equally well to BASIC for other computers.

NUMERIC AND STRING CONSTANTS

In terms of BASIC programming, a *constant* is a specific value; and as described in the following sections, constants can be classified as either *numeric constants* or *string constants*. Knowing how to work with numeric and string constants can eliminate a lot of undesirable and puzzling programming effects.

Numeric Constants

A numeric constant is some fixed numerical value. Here are a few numeric constants that are expressed in rather familiar forms:

23 - 212 1.6888 9.9999 1432000

Numeric constants can be very small or very large numbers, they can be positive or negative, and they can be whole numbers or numbers having decimal parts. Just ordinary number values—that is all numeric constants are.

There are a few special rules, limitations and forms of notation that apply to numeric constants in BASIC. For one, large numbers must not use commas in the conventional fashion; in fact they must not use commas at all. So if you wish to express a value of one million in BASIC, you must enter it as 1000000, and *not* as 1,000,000.

In keeping with the usual arithmetic convention, negative-valued numeric constants are preceded by a minus sign, while positive values can be expressed with a plus sign or no sign at all. If you enter a PRINT -128 command, for example, ATARI BASIC will respond by printing - 128 on the screen. But if you enter a PRINT +128 command, BASIC will exercise the no-sign option and print 128.

ATARI BASIC uses a number format called *floating-point notation*. Among other things, that means it expresses very large and very small values in terms of powers-of-10, or *scientific notation*. Very large numbers, in this context, are positive or negative numbers having more than nine digits to the left of the decimal point; and very small numbers are those between -0.01 and 0.01.

To see how ATARI BASIC handles very large values, enter this command:

```
PRINT 12345678901
```

and you will see this response:

```
1.2345678901E+10
```

The PRINT statement specifies a positive constant that has more than nine significant digits, and BASIC responds by converting the value to a form of scientific notation. The E + 10 in that response represents $\times 10^{10}$ —ten to the tenth power.

The same general idea applies to small numeric values. Enter this command:

```
PRINT 0.00012345
```

and ATARI BASIC will show this version:

```
1.2345E - 04
```

That value in the PRINT statement is less than 0.01, so ATARI BASIC automatically converts it to scientific notation, where E - 04 means 10^{-4} .

There are some limitations on the range of numeric values that can be expressed in ATARI BASIC, even when using scientific notation; and that range is:

– 9.99999999E + 97 through 9.99999999E + 97

Furthermore, there is a range of values that is very close to zero wherein a value will be automatically set to zero. That range is:

– 9.99999999E - 98 through 9.99999999E - 98

Finally, there are not only limitations on the values of numeric constants, but also on the number of significant nonzero digits—nine of them. ATARI BASIC allows only nine significant digits that are not zero; it accepts numbers having more than nine significant digits, but it sets all digits beyond the nine to zero.

The constant 123456789123456789, for example, has 18 significant digits, but ATARI BASIC will deal with it as 123456789000000000. There are still 18 significant digits, but only the first nine have nonzero values. And as described earlier, it will convert such values to scientific notation: 1.23456789E + 17.

The same principle applies to fractional numbers. A constant such as 0.00123456789123456789 has 20 significant digits to the right of the decimal point. Eighteen of them are nonzero digits, so ATARI BASIC will treat the number as 0.00123456789 and display it as 1.23456789E-03.

Summarizing the limitations and conventions required for numeric constants in ATARI BASIC:

- Large numeric values must not include commas.
- A negative sign (-) must precede a negative-valued constant, but a plus sign (+) is optional for positive-valued constants (ATARI BASIC will always print positive values *without* the plus sign).
- ATARI BASIC uses scientific notation for expressing very large and very small numeric constants.
- ATARI BASIC will deal with only nine nonzero significant digits; any beyond that number will be set to zero.

String Constants

Whereas numeric constants must be composed of meaningful numeric values, string constants can be constructed of any combination of letters, punctuation marks, special symbols and numerals. What's more, string constants usually must be enclosed in quotation marks.

The following BASIC statement prints a string constant, HELLO:

```
PRINT "HELLO"
```

The string constant in that instance is composed entirely of upper-case letters of the alphabet and, as is usually the case, the constant is enclosed in quotation marks. When you execute that PRINT command, however, you will see ATARI BASIC printing HELLO without the quotes.

As implied earlier, a string constant can be built from all sorts of letters and punctuation. For example:

```
PRINT "Hello, there, you silly goose!"
```

The string constant in that case includes both upper- and lower-case letters as well as some spaces and punctuation marks. The only punctuation that doesn't work is the quotation mark, itself; it is used to mark the beginning and end of a string constant, and, therefore, cannot appear within the string, itself. Most programmers cope with that little difficulty by using apostrophes where quotes would normally appear within a string.

Numerals that are included in a string constant are treated as literal characters rather than numeric values. You can prove that point with a simple demonstration. First, treat the expression $1 + 2$ as a string constant. In other words, execute this statement:

```
PRINT "1+2"
```

The fact that you have enclosed the expression within quotation marks means that the computer will interpret the numerals as part of a string, and it will respond by printing out a literal version of your string:

```
1+2
```

Next, execute this command:

```
PRINT 1+2
```

Omitting the quotation marks suggests that the numerals are to be treated as numeric constants, and the computer responds to that fact by printing the result of the summation operation:

```
3
```

The only limitation on the length of a string constant (the number of characters included in it) is the amount of available computer memory; however, it is a good practice to make a habit of dealing with 130 characters or less. The reason for that limit, incidentally, is that it is the size of the INPUT string buffer.

NOTE: There is a special string constant that contains no characters whatsoever. It is defined by a pair of successive quotation marks, "", and is called the null string.

Summarizing the rules and limitations for expressing string constants:

- In most instances, string constants must be enclosed in quotation marks. (The special exceptions will be described in later discussions.)
- String constants may be composed of any characters except a quotation mark.
- Numerals appearing within a string constant are treated as literal characters rather than numeric values.
- The recommended maximum length of a string constant is 130 characters.

BASIC VARIABLES AND VARIABLE NAMES

Most arithmetic and control operations in BASIC make reference to variables and, particularly, variable names. A *variable* is an expression that can take on a wide variety of different values—values that are assigned to variables through the normal execution of a program. A *variable name* is a set of one or more alphanumeric characters that you, the programmer, devise yourself according to a few simple rules.

NOTE: ATARI BASIC allows up to 128 different variables to be used through the execution of a program.

BASIC uses two kinds of variables: *numeric variables* and *string variables*. The following discussions point out their differences.

Numeric Variables

Numeric variables and numeric variable names refer to numbers or quantities. They are used in much the same way that variables are used in ordinary algebra. This 2-line program illustrates the use of a particular numeric variable:

```
10 AX=200
20 PRINT AX
```

Line 10 assigns a *numeric constant*, 200, to a numeric variable that has the name AX. Line 20 then prints the value that is currently assigned to variable AX—a value of 200 in this case. You could get the same overall result by executing:

```
PRINT 200
```

but that limits the operation to printing a single value. The advantage of using variables is that you can get this program to print some other number by assigning a different constant to AX in line 10. There is no need to adjust the PRINT statement in line 20, because that statement refers to the variable name in a general way and makes no specific reference to the constant value that is assigned at an earlier time.

A BASIC programmer has a great deal of latitude, and only a few rules to follow, when making up numeric variable names:

1. A numeric variable name is composed of upper-case letters of the alphabet and numerals 0 through 9; punctuation, including spaces and periods, is not allowed.
2. There has to be at least one valid character in a numeric variable name, of course, but the maximum number of characters is limited only by the ability to fit the name into the lines of BASIC programming that use it. (Generally, it is good practice to use shorter variable names, and yet make them long enough to be meaningful.)

3. Numeric variable names should not include the words shown in the Reserved Word List, Chart 2-1. (Although ATARI BASIC will usually accept variable names from the Reserved Word List, there are many instances where such names will cause an Error-interrupt during the execution of a program.)

Chart 2-1. ATARI BASIC Reserved Words List

ABS	IF	RESTORE
ADR	INPUT	RETURN
AND	INT	RND
ASC	LEN	RUN
ATN	LET	SAVE
BYE	LIST	SETCOLOR
CLOAD	LOAD	SGN
CHR\$	LOCATE	SIN
CLOG	LOG	SOUND
CLOSE	LPRINT	SQR
CLR	NEW	STATUS
COLOR	NEXT	STEP
COM	NOT	STICK
CONT	NOTE	STRIG
COS	ON	STOP
CSAVE	OPEN	STR\$
DATA	OR	THEN
DEG	PADDLE	TO
DIM	PEEK	TRAP
DOS	PLOT	USR
DRAWTO	POINT	VAL
END	POKE	XIO
ENTER	POP	
EXP	POSITION	
FOR	PRINT	
FRE	PTRIG	
GET	PUT	
GOSUB	RAD	
GOTO	READ	
GRAPHICS	REM	

NOTE: These words should not be used as variable names, nor should they appear within variable names. It is possible to break that rule in many instances, but a wise programmer will avoid the risks involved.

Here are some examples of valid numeric variable names:

TRY AXIS MODEL1 NAMESFROMTABLE100

And here are some *invalid* numeric variable names:

try	(uses lower-case words)
1TRY	(begins with a numeral)
DARLING.SET	(includes punctuation)
AND	(uses a reserved word)

String Variables

String variables and string variable names refer mainly to literal expressions, but they can also refer to combinations of special control operations and graphics symbols. This 3-line program illustrates the use of a particular string variable:

```
5 DIM AX$(10)
10 AX$="HELLO"
20 PRINT AX$
```

Line 5 dimensions a string variable, AX\$, for 10 characters, and then line 10 assigns a *string constant*, HELLO, to that variable. Line 20 prints the constant that is currently assigned to variable AX\$—HELLO in this case. You could get the same result on the screen by executing:

```
PRINT "HELLO"
```

but that limits the operation to printing a single string constant. The advantage of using string variables is that you can get this program to print some other string by assigning a different constant to AX\$ in line 10. There is no need to adjust the PRINT statement in line 20, because that statement refers to the variable name instead of a specific string constant that is assigned at an earlier time.

As with numeric variable names, a BASIC programmer has a great deal of latitude, and only a few rules to follow, when making up string variable names:

1. Every string variable name must end with a dollar-sign (\$) symbol.
2. A string variable name is composed of upper-case letters of the alphabet and numerals 0 through 9.

3. There has to be at least one valid character and a dollar sign in a string variable name, but the maximum number of characters is limited only by the ability to fit the name into the lines of BASIC programming that use it.
4. String variable names should not include the words shown on the Reserved Word List, Chart 2-1. (Like some numeric variable names, ATARI BASIC will often accept variable names from the Reserved Word List, even when appended with a dollar-sign symbol; but the risk of causing an Error-interrupt during the execution of a program makes it unwise to use such names.)

Here are some examples of valid string variable names:

TRY\$ AXIS\$ MODEL1\$ NAMESFROMTABLE10\$

And here are some *invalid* string variable names:

try\$	(uses lower-case letters)
1TRY\$	(begins with a numeral)
DARLING.SET\$	(includes punctuation)
MIX\$MONEY\$	(includes \$ as punctuation)
CHR\$	(uses a reserved word)
SALT	(does not end with \$)

NOTE: All string variables must be DIMensioned before they are used in a program. See DIMENSIONING String Variables and Numeric Arrays.

DIMENSIONING STRING VARIABLES AND NUMERIC ARRAYS

BASIC's DIM statement is always important for setting the DIMension of numeric and string variables. It is especially important in ATARI BASIC because it is required for establishing the maximum length of constants that are assigned to every string variable in the program. The DIM statement also sets up subscripted numeric variables and multidimensional numeric arrays. In spite of appearances to

the contrary, however, *ATARI BASIC* does not support subscripted string variables nor string arrays.

DIMensioning String Variables

Every string variable that is used in an *ATARI BASIC* program must be dimensioned with respect to the maximum number of characters they are to contain. The dimensioning operation must occur prior to using the string variables, and it should occur just one time through the execution of the program.

Suppose that you anticipate using a string variable `PY$` in a program, and you think that any constants assigned to it will contain no more than 10 characters. The appropriate dimensioning statement is:

```
DIM PY$(10)
```

Because the dimensioning statements must occur prior to any reference to the corresponding string variables, most programmers make it a habit of placing the `DIM` statements very early in the program—often in one of the very first lines. So it isn't at all unusual to see the first operational line in an *ATARI BASIC* program looking something like this:

```
10 DIM A$(1), X$(20), F$(6)
```

Having done that, string variable `A$` will work with 1 character, `X$` will work with up to 20 characters, and `F$` will work with as many as 6 characters.

The program will work, even if it happens that a dimensioned string variable picks up more characters than is set aside for it. In such cases, the computer simply *truncates*, or cuts off, the string after the dimensioned number of characters has been reached. The following program illustrates that point:

```
10 DIM NA$(5)
20 NA$="SOMETIMES"
30 PRINT NA$
```

Line 10 dimensions string variable NA\$ for a maximum of 5 characters, but line 20 assigns a string constant that has 9 characters in it. When line 30 prints the current string constant that is thus assigned to NA\$, it will show only the first 5 characters: SOMET.

In actual practice, it is often inconvenient, and sometimes impossible, to determine the maximum number of characters that will ever be assigned to a given string variable. The temptation in such instances is to go overboard, and dimension the variable at some large figure; say, DIM X\$(200). That should be avoided wherever possible, however, because the dimensioning operation reserves memory space for the string characters; and using needlessly large dimensioning values uses up memory that might be put to better use.

Incidentally, a CLR statement within a program clears all previous dimensioning specifications. That can, in effect, make it possible to redimension a string variable during the course of a program. But CLR makes it necessary to redimension *all* string variables to be used and sets the DATA pointer to the first item in the lowest-numbered data list. In other words, the notion of using a CLR statement to redimension a variable during the execution of a program is of questionable value.

Summarizing the procedures for dimensioning string variables:

- All string variables must be dimensioned with regard to the maximum number of characters they can handle.
- String variables must be dimensioned prior to using them, generally in one of the first lines of programming.
- String variables cannot be dimensioned more than one time during the execution of a program unless a CLR statement is first used to zero-dimension *all* previously dimensioned variables.

- Assigning a string constant that has more characters than is specified by the corresponding DIM statement simply truncates the string.

Subscripted Numeric Variables and Arrays

The DIM statement in ATARI BASIC serves two entirely different purposes for string and numeric variables. As described in the previous section, every string variable cited in a program must be DIMensioned according to the maximum number of characters expected for its string-constant assignments. Numeric variables, on the other hand, do not have to be dimensioned unless they are to be used as subscripted or multidimensional array variables. Whereas this simple string routine will not run without the benefit of the DIM statement in line 10:

```
10 DIM A$(1)
20 A$="G"
30 PRINT A$
```

the following numeric program runs quite well without any prior dimensioning of variable N:

```
10 FOR N=0 TO 100
20 PRINT N;
30 NEXT N
```

Subscripted numeric variables or arrays must be dimensioned prior to using them, however.

Generally speaking, subscripted numeric variables serve the same purpose as their counterparts in conventional algebraic notation. A mathematics or physics textbook, for example, might show some subscripted variables this way:

$$Y = x_1 + x_2 + x_3 + x_4$$

The idea is to indicate the sum of four different numeric variables that are closely related, but are able to take on different numeric-constant values. Because BASIC does not use subscripted characters, it is necessary to indicate those characters within parentheses. Here is the BASIC form of the “textbook” equation just cited:

$$Y=X(1)+X(2)+X(3)+X(4)$$

A program cannot refer to such subscripted numeric variables in ATARI BASIC without first dimensioning them with regard to the largest-value subscript you intend to use. So before it is possible to execute that BASIC statement, the program must include a DIM X(4) statement.

The general form of a subscript dimensioning statement is:

DIM *numvar*(*d*)

where *numvar* is any valid numeric variable name, and *d* is the largest subscript index to be used. Thus a typical line of programming for dimensioning several subscripted numeric variables might look like this:

```
10 DIM X(3), FE(20), YY(9)
```

That one will dimension subscripted variable X for *four* variables—X(0) through X(3). By the same token, it dimensions FE for subscripted variables FE(0) through FE(20), and YY for YY(0) through YY(9).

The following program prompts you to enter four different numbers. After that, it displays the four numbers, their sum, and their average. In this particular case, the numeric values are assigned to subscripted variables N(0) through N(3) by means of INPUT and assignment statements within a FOR . . . NEXT loop.

It can be very helpful to realize that the execution of a RUN command automatically sets any subscripted numeric values to zero. That eliminates the need for zeroing them as one of the first steps in a program.

```

10 DIM N (3)
20 FOR E=0 TO 3
30 PRINT "ENTER A NUMBER ";
40 INPUT X:N(E)=X
50 NEXT E
60 PRINT:PRINT
70 FOR E=0 TO 3
80 PRINT N(E);S=S+N(E)
90 NEXT E
100 PRINT "SUM=";S
110 PRINT "AVE=";S/4

```

A *numeric array* is an extension of subscripted variables. Instead of using just one subscripted index numeral, numeric arrays use two or more of them. The traditional foundation for arrays in BASIC are the matrices of modern algebra.

A 3×3 algebraic matrix is often organized this way in math-oriented books:

$$\begin{matrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{matrix}$$

The two subscripted numerals that are separated by a comma indicate the *row*, *column* locations. So variable $a_{2,3}$ indicates the third element in the second row.

Such elements are expressed in BASIC by including the index terms—also separated by a comma—within a set of parentheses. For example:

$$\begin{matrix} A(1,1) & A(1,2) & A(1,3) \\ A(2,1) & A(2,2) & A(2,3) \\ A(3,1) & A(3,2) & A(3,3) \end{matrix}$$

Those are examples of a 2-dimensional array for variable A; and that variable would have to be dimensioned as DIM A(3,3). The dimensioning operation for an array must indicate the largest value that is to appear in the corresponding element locations. A statement such as:

```
DIM XE(10,5)
```

would set aside space for 66 different array values for variable—row indices 0 through 10, and column indices 0 through 5.

The maximum size of an array is limited only by the amount of available RAM, but ATARI BASIC does not allow arrays of more than 2 dimensions.

It is particularly important to realize that *executing a RUN command in ATARI BASIC does not automatically set all elements in an array to zero*. It does so for simple numeric variables and subscripted numeric variables (1-dimensional arrays); but not for 2-dimensional arrays. So arrays not only have to be dimensioned early in the programming, but they must be initialized as well.

Suppose that a program is using a 2-dimensional array for variable G, where the row elements range from 0 through 4, and the column elements range from 0 through 8. A suitable initialization routine looks like this:

```
10 DIM G(4,8)
20 FOR X=0 TO 4:FOR Y=0 TO 8
30 G(X,Y)=0
40 NEXT Y:NEXT X
```

The DIM statement in line 10 dimensions the array. The remainder of the routine then uses nested FOR . . . NEXT loops to set all elements of the array (45 of them) to zero. Of course there has to be additional programming in order to make the operation meaningful, but at least this gets matters under control at the outset.

OPERATIONS AND OPERATORS

Much of the computing and control activity of a computer are determined by the nature of *operators* that are written into the programs. There are just three families of operators: arithmetic, relational, and logical operators. This section describes and compares those three families.

Arithmetic Operators

Most people who are attempting to learn something about computer programming are well aware of the application of *arithmetic operators*—special, well-defined symbols that indicate a mathematical operation that is to take place between two numeric constants or variables. Table 2-1 lists the arithmetic operators that apply under ATARI BASIC and, indeed, most other versions of BASIC.

Table 2-1. BASIC's Arithmetic Operators

Operator	Function
+	addition (sum)
-	subtraction (difference)
*	multiplication (product)
/	division (quotient)
^	exponentiation (power)
-	negation (change of sign)

It is easy to check out the function of these operators by entering some simple commands that use them. The computer, in effect, works like a calculator.

PRINT 2 + 3 will show the sum, 5, on the screen.

PRINT 2 - 3 will show the difference, -1, on the screen.

PRINT 2*3 will show the product, 6, on the screen.

PRINT 3/2 will show the quotient, 1.5, on the screen.

PRINT 3^2 will evaluate 3^2 and show 9 on the screen.

PRINT 2^3 will evaluate 2^3 and show 8 on the screen.

X = 2:PRINT -X will negate the value originally assigned to variable X, and show -2 on the screen.

Relational Operators

Relational operators suggest the relative magnitudes of numeric values or values assigned to numeric variables. The operations apply equally well to string values and variables, but their applications in that case are rather different. In either case, the operators are generally meaningful only when used in conditional, IF . . . THEN statements. Table 2-2 shows BASIC's family of relational operators.

Table 2-2. BASIC's Relational Operators

Operator	Function
<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	does not equal

Like the arithmetic operators, most of the relational operators look like their counterparts in most math-oriented text. The *not-equal* is the only one that is different; it is most often shown as \neq in noncomputer literature.

The following programming routines let you observe the function of BASIC's relational operators.

1. The *less-than* operator:

```
10 X=1
20 IF X < 10 THEN 40
30 END
40 PRINT X;
50 X=X+1
60 GOTO 20
```

The *less-than* conditional statement in line 20 allows the program to print the current value of X only as long as that value is less than 10. The program, in other words, prints integers 1 through 9, inclusively.

2. The *greater-than* operator:

```
10 X=1
20 IF X > 9 THEN END
30 PRINT X;
40 X=X+1
50 GOTO 20
```

That program also prints integer values of X in the range of 1 through 9. Line 20, however, uses a *greater-than* operator to determine when the current value of X exceeds 9 and, consequently, brings the program to an end.

3. The *equal-to* operator:

```
10 X=1
20 IF X=10 THEN END
30 PRINT X;
40 X=X+1
50 GOTO 20
```

The routine prints integers 1 through 9, and comes to an end when the *equal-to* condition in line 20 is satisfied.

4. The *less-than-or-equal-to* operator:

```
10 X=1
20 IF X < 10 THEN 40
30 END
40 PRINT X;
50 X=X+1
60 GOTO 20
```

This is yet another way to print integers from 1 through 9. In this case, the program continues printing those integers as long as X is less than 10 as determined by the conditional statement in line 20.

5. The *greater-than-or-equal-to* operator:

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

The end of the integer-printing operation is detected by the *greater-than-or-equal-to* operator in line 20.

6. The *does-not-equal* operator

```
10 X=INT(10*RND(1))
20 PRINT "GUESS A NUMBER BETWEEN 0 AND 9 ";
30 INPUT N
40 IF N <> X THEN PRINT "NOPE":GOTO 20
50 PRINT "THAT'S IT!"
```

Line 10 assigns a randomly generated integer value in the range of 0 through 9 to variable X. Lines 20 and 30 then prompt you to enter your guess at that number and, subsequently, assign it to variable N. Line 40 compares the current values of N and X; if they are *not* equal, then the program prints NOPE and loops back to line 20 to give you another chance. But if it turns out that N is indeed equal to X, the *does-not-equal* condition in line 40 fails, and the program concludes at line 50 by printing THAT'S IT.

The foregoing examples all refer to relational operations as they apply to numeric constants and variables. They apply equally well to string constants and variables, however.

The notion of using relational operators to test relationships between strings might seem puzzling at first to anyone who has very little experience with BASIC. It might seem curious, for example, to question whether HELLO is greater-than-or-equal-to GOOD BYE—but it does make sense.

One way to think of relational operations between strings is in terms of alphabetical sequence. And in that context, it can be helpful to interpret the relational operator this way:

$A\$ < B\$$ —the string currently assigned to $A\$$ occurs alphabetically before the one assigned to $B\$$.

$A\$ > B\$$ —the string currently assigned to $A\$$ occurs alphabetically after the one assigned to $B\$$.

$A\$ = B\$$ —the string currently assigned to $A\$$ is alphabetically identical to the one assigned to $B\$$.

$A\$ \leq B\$$ —the string currently assigned to $A\$$ occurs alphabetically before, or is identical to, the one assigned to $B\$$.

$A\$ \geq B\$$ —the string currently assigned to $A\$$ occurs alphabetically after, or is identical to, the one assigned to $B\$$.

$A\$ <> B\$$ —the string currently assigned to $A\$$ is somehow different from the one assigned to $B\$$.

Here is one of the most popular applications of string comparisons:

```
10 DIM K$(1)
20 PRINT "DO YOU WANT TO QUIT (Y/N)";
30 INPUT K$
40 IF K$="Y" THEN END
50 GOTO 20
```

The basic idea is to determine whether or not the user wants to end an ongoing program. Line 20 prompts the user to enter a Y or N response, and line 30 assigns that string response to variable $K\$$. The conditional statement in line 40 then tests the user's response against constant Y; if, indeed, the operator's response is Y, then the program ends. Otherwise it loops back to line 20 to prompt the user once again.

The next program lets the user enter two different words, prints them in alphabetical order, and asks whether or not the user wants to do the whole operation again.

```
10 DIM K$(1),X$(10),Y$(10)
20 PRINT "ENTER A WORD ";
30 INPUT X$
40 PRINT "ENTER A SECOND WORD ";
50 INPUT Y$
60 IF X$ >= Y$ THEN PRINT Y$:PRINT X$:GOTO 80
70 PRINT X$:PRINT Y$
80 PRINT:PRINT "DO AGAIN (Y/N)";
90 INPUT K$
100 IF K$="N" THEN END
110 PRINT:PRINT:GOTO 20
```

Program lines 20 through 50 prompt the user to enter two words and, in the process, the statements assign those words to X\$ and Y\$. The conditional statement in line 60 compares the two words; and if X\$ occurs later alphabetically, or is identical to Y\$, the program prints Y\$ followed by X\$. But if the relational condition in line 60 is *not* satisfied, the program executes line 70 to print X\$ followed by Y\$.

As mentioned earlier in this discussion, there is something more to using strings with relational operators than looking at alphabetical arrangements. BASIC actually looks at the strings in terms of the ATASCII code number for each character. (ATASCII is an acronym for ATari ASCII; and ASCII is an acronym for the standard character-coding format, American Standard Code for Information Interchange.) Table 2-3 shows the ATASCII code numbers for the characters most relevant to this discussion; and a complete listing appears in Appendix B.

Table 2-3. Partial Listing of Characters and ASCII Codes

ASCII	Char	ASCII	Char	ASCII	Char
32	(space)	63	?	93]
33	!	64	@	94	.
34	"	65	A	95	_
35	#	66	B	96	\
36	\$	67	C	97	a
37	%	68	D	98	b
38	&	69	E	99	c
39	'	70	F	100	d
40	(71	G	101	e
41)	72	H	102	f
42	*	73	I	103	g
43	+	74	J	104	h
44	,	75	K	105	i
45	-	76	L	106	j
46	.	77	M	107	k
47	/	78	N	108	l
48	0	79	O	109	m
49	1	80	P	110	n
50	2	81	Q	111	o
51	3	82	R	112	p
52	4	83	S	113	q
53	5	84	T	114	r
54	6	85	U	115	s
55	7	86	V	116	t
56	8	87	W	117	u
57	9	88	X	118	v
58	:	89	Y	119	w
59	;	90	Z	120	x
60	<	91	[121	y
61	=	92	\	122	z
62	>				

NOTE: See Appendix B for a complete listing.

According to that table, the upper-case letter A has a lower ATASCII code number than does the upper-case Z. It is the code number that is used when a BASIC statement determines whether one letter of the alphabet should appear before another. That principle leads to the fact that BASIC will regard all numerals as being "earlier in the alphabet" than all other letters are. In other words, a string version of numeral 9 is less than A. By the same token, all lower-case letters carry larger ATASCII-code values than the upper-case letters do; thus, a will be regarded as appearing "later" in the alphabet than Z does.

That fact causes some difficulty when you are attempting to write a program that sorts strings into alphabetical order when they are composed of combinations of upper- and lower-case characters. (The way around the difficulty is to write a routine that converts all lower-case letters to their upper-case counterparts—by subtracting a value of 32 from all lower-case ATASCII code numbers.)

Logical Operators

Table 2-4 shows the logical operators that are directly available from ATARI BASIC. There are just three of them—AND, OR, and NOT—but they are sufficient to perform any desired logical operation.

Table 2-4. ATARI BASIC's Logical Operators

Operator	Function
AND	Logical AND
OR	Logical OR
NOT	Logical negation

Used in conjunction with IF . . . THEN conditional statements, the AND and OR operators connect two or more expressions in explicit ways. Consider this general type of AND statement:

IF *expr1* AND *expr2* THEN *expr3*

The literal interpretation of that statement is: IF *expr1* is true AND if *expr2* is also true, THEN execute *expr3*. IF either *expr1* or *expr2* is not true, then the implication is that computer operations should ignore the action prescribed by *expr3*, and go to the next statement in the program.

By way of a specific example:

IF A >=0 AND A <=9 THEN PRINT A

When executing that statement, the computer will print the value currently assigned to variable A only if that value is between 0 and 9, inclusively.

A conditional statement that uses AND operators is satisfied only when *all* of the ANDed expressions are true at the same time.

Consider this general form of a conditional statement that uses an OR operator:

IF *expr1* OR *expr2* THEN *expr3*

The literal interpretation of that statement is: IF *expr1* is true, if *expr2* is true, OR if both are true, THEN execute *expr3*. IF both *expr1* and *expr2* are not true, then the implication is that computer operations should ignore the action prescribed by *expr3*, and go to the next statement in the program.

Here is a specific example:

IF A < 0 OR A > 9 THEN END

When executing that statement, the program will end if the value currently assigned to variable A is less than 0 or if it is greater than 9.

A conditional statement that uses OR operators is satisfied when one or more of its ORed expressions are true.

The AND and OR operators are both *binary* logical operators in the sense that they connect at least two different expressions. The NOT operator, on the other hand, is an *unary* operator—it applies to a single expression.

The NOT operator reverses the logic of the expression to which it applies. For instance:

IF NOT A > 10 THEN END

That statement literally says: IF the value currently assigned to variable A is NOT greater than 10, then END the program.

The foregoing examples each used one particular logical operator; but it is often necessary to use various combinations of them within the same conditional statement.

Order of Precedence for Operators

Many programming situations call for including more than one operator within a statement—combinations of more than one arithmetic, relational, and logical operator. That situation gives rise to a question concerning the order of precedence. Will the operators be regarded by the computer in a strict left-to-right sequence, or will some operators take a priority of execution over others? The answer to that question is, "Yes."

Table 2-5 shows the *order of precedence* for the execution of BASIC's operators. Those having a higher order of precedence will always be executed before others having a lower order. And operators having the same order of the precedence will be executed in a left-to-right sequence.

**Table 2-5. Order of Precedence of
ATARI BASIC Operators**

Operator	Function	Precedence Level
Grouping ()	Sign of grouping	9 Highest precedence
Arithmetic Operators		
^	Exponentiation	8
-	Negation	7
*	Multiplication	6
/	Division	6
+	Addition	5
-	Subtraction	5
Relational Operators		
=	Equal	4
<	Less than	4
>	Greater than	4
<=	Less than or equal to	4
>=	Greater than or equal to	4
<>	Not equal	4
Logical Operators		
NOT	Logical negation	3
AND	Logical AND	2
OR	Logical OR	1 Lowest precedence

Suppose that you have written a BASIC statement of this form:

$$A=B+2-C/16*D$$

Of all the operators appearing in that statement, the multiplication operator has the highest priority. The significance of that fact is that the computer will do the $16*D$ operation before any others. It will then deal with the division operator; having already multiplied constant 16 by variable D, the computer divides that result into variable C. What remains, then, are the addition and subtraction operations.

According to the established order of precedence, addition and subtraction operators have the same order of precedence; and in such instances, the computer executes the operators from left to right. In this particular example, that means summing variable B with constant 2, and then subtracting the result of the previously executed $C/16*D$ operators from the result.

The same idea applies to all of the operators cited in the table. Notice that all of the relational operators have the same level of precedence. That means they will be regarded in a strict left-to-right sequence. The logical operators are at the bottom of the list, thereby taking the lowest levels of precedence.

Perhaps it seems that a need for setting up BASIC statements according to those fixed orders of precedence makes it troublesome for a programmer. That isn't quite the case, however. BASIC includes signs of grouping, parentheses, that take precedence over all other operators. And that means it is possible to specify the operators in any convenient or meaningful sequence and then use sets of parentheses to establish the order of execution.

Recall this BASIC statement:

$$A=B+2-C/16*D$$

It is possible to avoid any troublesome references to the orders of execution by expressing it this way:

$$A=B+2-(C/(16*D))$$

Knowing that *nested* parenthetical expressions are always executed from the innermost to outermost levels, that form of the statement more clearly suggests: multiply 16 times D, divide the result into C, subtract the result from 2 and add variable B. Having to recall the orders of precedence is far less important when using parentheses to dictate the sequence.

Chapter 3

The BASIC Programming Language

This chapter summarizes the ATARI BASIC programming language. Many of the examples apply equally well to other versions of BASIC as run on other kinds of personal computers. But for the most part, the examples and discussions assume that you are working with the ATARI BASIC cartridge.

The purpose of the chapter is to provide a ready reference for the BASIC commands, statements, and functions. There are some differences between ATARI BASIC and some other commonly used versions; and readers who have previously learned BASIC from other sources will find this material a handy guide for dealing with those differences.

It is beyond the scope of this book to deal with the general principles of BASIC programming on an elementary level, however; so anyone who is not already acquainted with the essential elements of programming in BASIC should refer to an appropriate beginner's book.

A SUMMARY OF STATEMENTS, COMMANDS, AND FUNCTIONS

Chart 3-1 is an alphabetical listing of the primitive commands, statements, and functions for ATARI BASIC. Most of them will be familiar to readers who have learned BASIC before, and they can be found in most other books that deal with BASIC in a general fashion.

Table 3-1 classifies the instructions according to their general use and, incidentally, cites some abbreviated forms that can simplify the keyboard-programming task.

Generally, the *BASIC Commands* are executed directly from the keyboard while the computer is in its program, or *immediate*, mode of operation. Most of them, however, can be included within a program and, in fact, are truly useful only in that context.

The *Control Statements* control the flow of an operating BASIC program. As shown in the table, many of them represent two-word statements.

The *Input/Output Commands* generally control the flow of information—into the computer, out of the computer, and between different sections within the computer.

There are relatively few ATARI BASIC commands that are specifically designed for string operations; but the *String Commands and Functions* can be used with other statements to yield a wide range of useful string manipulations.

The *Math Functions* are common to most versions of BASIC and they can be organized to execute virtually any mathematical calculation. A table of Derived Trigonometric Functions appears in Appendix F.

There are only two *Array Statements*; however, it is quite possible to compound other kinds of statements to manipulate any sort of array or matrix.

**Chart 3-1. Alphabetical Listing of ATARI BASIC
Primitive Commands, Statements, and Functions**

ABS	NEXT
ADR	NOTE
ASC	ON
ATN	OPEN
BYE	PADDLE
CLOAD	PEEK
CHR\$	PLOT
CLOG	POINT
CLOSE	POKE
CLR	POP
COLOR	POSITION
COM	PRINT
CONT	PTRIG
COS	PUT
CSAVE	RAD
DATA	READ
DEG	REM
DIM	RESTORE
DOS	RETURN
DRAWTO	RND
END	RUN
ENTER	SAVE
EXP	SETCOLOR
FOR	SGN
FRE	SIN
GET	SOUND
GOSUB	SQR
GOTO	STATUS
GRAPHICS	STEP
IF	STICK
INPUT	STRIG
INT	STOP
LEN	STR\$
LET	THEN
LIST	TO
LOAD	TRAP
LOCATE	USR
LOG	VAL
LPRINT	XIO
NEW	

**Table 3-1. ATARI BASIC Commands, Statements,
and Functions
Arranged by Their General Purposes**

ATARI BASIC Commands	
Command	Abbreviation
BYE	B.
CONT	CON.
END	none
LET	LE.
LIST	L.
NEW	none
REM	R. or period followed by a space
RUN	RU.
STOP	STO.

ATARI BASIC Control Statements	
Command	Abbreviation
FOR	F.
GOSUB	GOS.
GOTO	G.
IF	none
NEXT	N.
ON	none
POP	none
RESTORE	RES.
RETURN	RET.
STEP	none
THEN	none
TO	none
TRAP	T.

NOTE: Some of these statements must be used in conjunction with others in order to complete their meaning:

FOR ... TO ... NEXT
 GOSUB ... RETURN
 ON ... GOSUB
 FOR ... TO ... STEP ... NEXT
 IF ... THEN
 ON ... GOTO

**Table 3-1—cont. ATARI BASIC Commands,
Statements, and Functions
Arranged by Their General Purposes**

Input/Output Commands	
Command	Abbreviation
CLOAD	CLOA.
CLOSE	CL.
CSAVE	CS.
DATA	D.
DOS	DO.
ENTER	E.
GET	GE.
INPUT	I.
LOAD	LO.
LPRINT	LP.
NOTE	NO.
OPEN	O.
POINT	PO.
PRINT	PR. or ?
PUT	PU.
READ	REA.
SAVE	S.
STATUS	ST.
XIO	X.

NOTE: Some of these statements must be used in conjunction with another in order to make up complete I/O operations:

OPEN . . . CLOSE
 READ . . . DATA
 OPEN . . . END

ATARI BASIC String Commands and Functions	
Command	Abbreviation
ASC	none
CHR\$	none
LEN	none
STR\$	none
VAL	none

**Table 3-1—cont. ATARI BASIC Commands,
Statements, and Functions
Arranged by Their General Purposes**

ATARI BASIC Math Functions	
Function	Abbreviation
ABS	none
ADR	none
ATN	none
CLOG	none
COS	none
DEG	none
EXP	none
FRE	none
INT	none
LOG	none
PEEK	none
POKE	none
RAD	none
RND	none
SGN	none
SIN	none
SQR	none
USR	none

ATARI BASIC Array Statements	
Statement	Abbreviation
DIM	DI.
CLR	none

ATARI BASIC Graphics Commands	
Command	Abbreviation
GRAPHICS	GR.
COLOR	C.
DRAWTO	DR.
GET	GE.
LOCATE	LOC.
PLOT	PL.
POSITION	POS.
PUT	PU.
SETCOLOR	SE.
XIO	X.

**Table 3-1—cont. ATARI BASIC Commands,
Statements, and Functions
Arranged by Their General Purposes**

ATARI BASIC Sound and Game-Controller Commands	
Command	Abbreviation
SOUND	SO.
PADDLE	none
PTRIG	none
STICK	none

The *Graphics Commands* refers to what is now commonly regarded as the most powerful feature of the ATARI system—its graphics.

The *Sound and Game-Controller Commands* might be better classified among the I/O commands; but they are set apart here in order to underscore their existence as a special kind of command.

A SUMMARY OF ATARI BASIC SYNTAX AND APPLICATIONS

This section describes, in alphabetical order, all ATARI BASIC commands, statements and functions. The purpose is to show the proper syntax and suggest some applications. Most of the descriptions are complete in their own right, but others refer you to further details in other chapters.

ABS(x)

ABS(x) is a numeric function that returns the absolute, or unsigned, value of x, where x is any numeric constant or expression.

Examples:

```
PRINT ABS(-1.2),ABS(30)
```

The example will print 1.2 and 30—the unsigned values of -1.2 and 30.

```

10 FOR N = -5 TO 5
20 PRINT N,ABS(N)
30 NEXT N

```

That program will return this sort of display:

```

- 5    5
- 4    4
- 3    3
- 2    2
- 1    1
  0    0
  1    1
  2    2
  3    3
  4    4
  5    5

```

ADR(x\$)

ADR(x\$) is a special control function that returns the RAM address of the first character in string **x\$**; where **x\$** is a string variable or constant.

Whenever a program dimensions a string variable, ATARI BASIC sets aside the specified amount of RAM space. This control function allows you to locate the starting address of that string. The function is especially useful for passing string variables to USR—called machine-language routines (see Chapter 7).

Demonstration Program:

```

10 DIM M$(5)
20 M$="HELLO"
30 SA=ADR(M$)
40 FOR I=0 TO LEN(M$)-1
50 PRINT SA+I,CHR$(PEEK(SA+I))
60 NEXT I

```

Lines 10 and 20 simply dimension string variable M\$ and assign string HELLO to it. Line 30 uses the ADR function to determine the starting address of the string and assigns that value to SA. The remaining lines then increment the address locations, PEEKing into each one and printing the addresses and character contained in each one of them.

AND

AND is used as a logical operator in conditional statements such as:

IF *expr1* AND *expr2* THEN *statement*

The *statement* is executed only if both *expr1* AND *expr2* are true.

Examples:

IF $X \geq 0$ AND $X \leq 9$ THEN PRINT X

The statement will print the current value of X only if that value is both greater than or equal to 0 AND less than or equal to 9. If X happens to be less than 0 or greater than 9, the PRINT *statement* is not executed.

IF K\$="Y" AND V < 10 THEN END

That line will END its program only if string K\$ is Y AND the value currently assigned to V is less than 10.

See Chapter 2 for further details.

ASC(x\$)

ASC(x\$) is a string function that returns the ATASCII code number for the first character in string x\$; where x\$ is a string variable or constant. If x\$ is the null character, the function returns an ATASCII value of 44.

Examples:

```
10 DIM M$(1)
20 M$="Y"
30 PRINT ASC(M$)
```


That routine will print 89 to the screen—the ATASCII code for upper-case Y.

```
10 DIM M$(5)
20 M$="HELLO"
30 PRINT ASC(M$)
```

In this instance, the computer will print 72 to the screen—the ATASCII code for the first character in HELLO.

On occasions where it is necessary to print the ATASCII codes for *all* of the characters in a specified string variable, use this sort of routine:

```
10 DIM M$(5)
20 M$="HELLO"
30 FOR C=0 TO LEN(M$)-1
40 PRINT PEEK(ADR(M$)+C)
50 NEXT C
```

ATN(x)

ATN(x) is a numeric function that returns the Arctangent, or inverse tangent, of x; where x is any floating-point constant or expression.

The Arctangent value of a number is the angle whose tangent is equal to that number. BASIC will normally express the angle in units of radian measure, but preceding the operation with a DEG command will cause all angles to be expressed in degrees.

BYE

Abbreviation: B.

BYE is a command that switches from ATARI BASIC to the Memo Pad screen mode. The command does not affect resident BASIC programming, current variable assignments, DOS or RS-232 assignments. Return to BASIC from the Memo Pad by striking the SYSTEM RESET key.

CLOAD

Abbreviation: CLOA.

CLOAD is a command that loads tokenized BASIC into the computer from the program recorder. Upon executing the command, the computer will generate a brief audio signal. Respond to that signal by locating the beginning of the program to be loaded, depressing the PLAY switch on the recorder and striking any key on the console (except the BREAK key). The loading is done when the READY message is printed again to the screen.

CLOAD works only with cassette-based programs that have been recorded with the tokenizing CSAVE or SAVE statements. It does not work with ATASCII-formatted files that have been saved by means of the LIST command. See Chapter 6 for further information.

CLOAD operations automatically refer to the device to IOCB Channel 7, so it must be closed if it has been opened for any other purpose.

CHR\$(x)

CHR\$(x) returns the string value that is represented by ATASCII code x; where x is a numeric constant or expression that has a value between 0 and 255.

Examples:

```
10 FOR C=97 to 122
20 PRINT C,CHR$(C)
30 NEXT C
```

The routine prints the ATASCII codes and characters for all lower-case letters of the alphabet.

```
10 FOR C=97 to 122
20 CHR$(C),CHR$(C-32)
30 NEXT C
```

This example increments through the ATASCII codes for the lower-case letters of the alphabet. Line 20 first prints the lower-case character and then subtracts 32 from the current ATASCII value to specify and print the upper-case version.

CLOG(x)

CLOG(x) is a numeric function that returns the base-10 logarithm of x ; where x is a floating-point constant or numeric expression that is greater than zero.

Use the LOG(x) function when it is desirable to obtain the natural, base-e logarithm of x .

CLOSE #x

Abbreviation: CL.

CLOSE is a statement that closes the IOCB channel that is specified by x ; where x is a valid channel number—an integer value between 0 and 7.

Executing an END statement closes all IOCB channels except 0, which is normally assigned to the screen editor.

See Chapter 6 for further details concerning IOCB operations.

CLR

CLR is a command that sets all numeric variables and DIM assignments to zero, and resets the DATA-list pointer.

An inappropriate application of CLR can destroy vital information; but a proper application makes it possible to zero numeric arrays very efficiently, redimension string variables, and re-READ a DATA list from the beginning.

COLOR x

Abbreviation: C.

COLOR is a graphics function that determines the color register for subsequent PLOT and DRAW TO statements; where x is a positive numeric value or expression.

The relationship between the COLOR value and the color of the text or graphic printed on the screen depends largely upon the current graphics mode. See more details in Chapter 4.

COM x\$(x),y\$(y) . . .
COM x1(y1,z1), x2(y2,z2) . . .

The COM statement defines the length of string variables and the size of numeric arrays. Its application is virtually identical to the more commonly used DIM statement.

Example:

COM SA(4,5),M\$(5)

That command dimensions numeric variable SA for a 4 by 5 array and sets the length of string variable M\$ to 5. It is functionally identical to:

DIM SA(4,5),M\$(5)

CONT

Abbreviation: CON.

CONT is a command that can, under the proper circumstances, allow a program to CONTINUE after it has been halted. Normally it is used in conjunction with the STOP command as a program-debugging tool—whenever program execution is halted by a STOP command, you can resume operations from the beginning of the following program line by entering the CONT command.

CONT can also be used in conjunction with a BREAK-key depression to stop and then resume the LISTing of a program on the video screen. Once a LIST is initiated, it can be stopped for inspection by striking the BREAK key; and then the listing can be resumed by entering the CONT command.

The CONT command does not yield reliable results when it is used after other kinds of program-halting events have occurred.

COS(x)

COS(x) is a numeric function that returns the trigonometric cosine of an angle expressed as x ; where x is a floating-point constant or numeric expression. The expression, x , is normally taken as an angle that is expressed in radians, but it can be expressed in degrees by preceding the functions with a DEG command.

CSAVE

Abbreviation: CS.

CSAVE is a command that is used for saving tokenized BASIC programs on the program recorder. Upon executing the command, the computer will respond with two beeping signals. Begin the recording by depressing the PLAY and RECORD buttons on the program recorder and striking any key on the console, except the BREAK key. The system signals the end of the saving operation by reprinting the READY message.

CSAVE automatically opens IOCB Channel 7, so it is important that the channel be closed prior to executing the CSAVE command.

Because CSAVE saves BASIC programs in a tokenized format, they can be reloaded only by means of the CLOAD command. See Chapters 7 and 8 for further details about tokenized BASIC.

DATA list

Abbreviation: D.

DATA is a nonexecutable statement that contains the *list* of items to be read by a READ command. The items in the *list* can be numeric constants or string constants separated by commas. String constants in a DATA list need not be enclosed in quotation marks.

Numerical and string constants can be mixed within the same DATA *list*, but only as long as the corresponding READ operation specifies the correct variable type.

Examples:

```
10 FOR N=1 TO 5
20 READ A: PRINT A
30 NEXT N
40 DATA 100,200,300,400,500
```

That routine will read and print the numeric constants, one at a time, from the DATA list.

```
10 DIM A$(10)
20 FOR N=1 TO 5
30 READ A$:PRINT A$
40 NEXT N
50 DATA, HARRY,GEORGE,RALPH,CINDY,JUDY
```

The routine reads and prints the five string constants specified in the DATA list.

```
10 DIM N$(10)
20 FOR N=1 TO 4
30 READ N$,A:PRINT N$,A
40 NEXT N
50 DATA SUSAN,20,JENNIFER,18,MIKE,22,TED,21
```

Line 30 reads a string value followed by a numeric value, and then prints them to the screen in that sequence. Notice that the DATA listing includes variable types in that same sequence—a string followed by a numeric value.

See the READ and RESTORE statements for further examples and relevant details.

DEG

Abbreviation: DE.

DEG is a command that converts all subsequent angular expressions to units of degrees. Use the RAD command to reset the normal units of radians.

DIM x\$(x),y\$(y) . . .

DIM x1(y1,z1), x2(y2,z2) . . .

Abbreviation: DI.

The DIM statement defines the length of string variables and the size of numeric arrays. It must be executed in the program prior to making any reference to the variables.

Examples:

DIM M\$(20)

That statement dimensions string variable M\$ for up to 20 characters.

DIM X(4)

That statement dimensions five subscripted variables of X—variables X(0) through X(4).

DIM M\$(10),X(2,4)

That statement dimensions string M\$ for up to 10 characters, and defines a 2 by 4 array for variable X.

See Chapter 2 for further details about dimensioning string variables, subscripted numeric variables, and numeric variable arrays.

DOS

DOS is a command that brings up the utility menu for the Disk Operating System (see Chapter 1). Return to BASIC by selecting menu-item B or striking the SYSTEM RESET key.

DRAWTO x,y

Abbreviation: DR.

DRAWTO draws a straight line from the last-plotted graphics point to a point having coordinates *x* and *y*; where *x* and *y* are numeric constants or expressions that evaluate to valid points for the current graphics mode.

See further details in Chapter 4.

END

END is a statement that is sometimes necessary for signaling the final statement in a BASIC program. The statement is not necessary if the last statement to be executed in the program happens to carry the highest-valued line number.

END closes all IOCB channels except No. 0, which is normally assigned to the screen editor.

ENTER *device*

Abbreviation: E.

ENTER is a command that is used for merging ATASCII-coded BASIC programming from *device* (cassette or disk) to the computer's memory. Existing programming is not erased; rather, the new program lines are added to (or merged with) the existing ones. In instances where the new programming has line numbers that are identical to those already in program RAM, the new numbers and statements write over the old ones.

In a similar way, ENTER can add new variable specifications to the existing program, and it will not affect existing specifications unless they are redefined by the incoming program.

ENTER automatically opens IOCB Channel 7 for loading the new programming, and it closes that channel when the transfer is completed.

Because ENTER works only with ATASCII-coded programming, it loads only tokenized BASIC programs that were saved by means of the LIST command.

Examples:

ENTER "C:"

That command loads, or merges, untokenized files from the program recorder.

ENTER "D:NEWSTF.ASC"

That command merges an ATASCII-coded program named NEWSTF.ASC from the default disk.

See Chapter 6 for further details regarding the nature of tokenized programs and nontokenized files.

EXP(x)

EXP(x) is a numeric function that returns constant e (2.71828182) to a power of x ; where x is a floating-point numeric constant or expression.

FOR var = *begin* TO *end* [STEP y]

Abbreviations: FOR F.
 NEXT N.

This statement defines and initiates a FOR . . . NEXT loop. A numeric variable name, *var*, is stepped between values *begin* and *end*; where *begin* and *end* are numeric constants or expressions. Unless specified otherwise by the STEP option, the FOR . . . NEXT loop increments *var* in steps of +1. Other STEP intervals are determined by the numeric constant or expression, *y*.

FOR must be used with a corresponding NEXT statement.

Examples:

```
10 FOR N=0 TO 9
20 PRINT "HELLO"
30 NEXT N
```

That example prints HELLO 10 consecutive times on the screen.

```
10 FOR X=0 TO 100 STEP 2
20 PRINT X
30 NEXT X
```

That example prints even-valued integers between 0 and 100.

```
10 FOR DWN=10 TO -10 STEP -1
20 PRINT DWN
30 NEXT DWN
```

That routine prints backwards from 10 to -10, and prints each of the integers in that range.

```
10 FOR N=0 TO 9
20 PRINT N;
30 FOR T=0 TO 100
40 NEXT T
50 NEXT N
```

The example uses an *outer loop* to establish and print values of N and a second, *inner loop* to execute a time delay between each printing. The example, in other words, uses *nested FOR . . . NEXT* loops. Notice that the FOR statements refer first to variable N and then T, while the NEXT statements refer to variable and then N.

There are a couple of techniques for breaking out of a FOR . . . NEXT loop before *var* reaches the *end* value. One way is to assign a value to *var* that is equal to *end* within the loop, itself. Basically, the following program counts and prints integer values from zero to whatever value is assigned to variable N, but never more than 9. So if you respond to the prompting message by entering a 4, the program will print integers 0 through 4; if you enter 9, it will print 0 through 9; but if you enter any number larger than 9—even 1000000—line 50 forces the FOR . . . NEXT loop to a premature conclusion.

```

10 PRINT "ENTER A POSITIVE NUMBER:";
20 INPUT N
30 FOR K=0 TO N
40 PRINT K
50 IF K >=9 THEN K=N
60 NEXT K

```

A second way to terminate a FOR . . . NEXT loop is to use statements POP:GOTO *line number*. The following example runs just like the previous one, but uses the POP statement to force the computer to “forget” it is running a FOR . . . NEXT loop. A NEXT-Without-FOR error will occur if you fail to follow the POP statement with a well-defined GOTO statement.

```

10 PRINT "ENTER A POSITIVE NUMBER:";
20 INPUT N
30 FOR K=0 TO N
40 PRINT K
50 IF K >=9 THEN POP:GOTO 70
60 NEXT K
70 END

```

FRE(x)

FRE(x) is a numeric function that returns the number of bytes of RAM that remain available for the user’s purposes. Variable x is a “dummy variable,” so its value is not relevant as long as it is a valid floating-point constant or numeric expression.

Example:

```
PRINT FRE(0)
```

That will print to the screen the amount of RAM that remains available for the program.

GET #x, var

Abbreviation: GE.

GET fetches a single byte from opened IOCB Channel *x*, and subsequently assigns the value to *var*. The type (numeric or string) of the fetched item must match that of *var*.

Example:

```
10 OPEN #1,4,0,"K:"  
20 GET #1,K$  
30 IF K$ <="A" AND K$ <="Z" THEN PRINT K$;  
:GOTO 20
```

Line 10 opens IOCB Channel 1 for input from the console keyboard. Line 20, in effect, waits for a single keystroke; and when it occurs, it assigns the character string to variable *K\$*. If, according to line 30, the character is an upper-case letter of the alphabet, the program prints the character and loops back to line 20 to fetch the next keystroke. A keystroke representing anything but an upper-case letter will bring the program to an end (and close IOCB Channel 1). That particular example can be quite useful for fetching single-keystroke characters without having to conclude the entry by striking the **RETURN** key (as is the case with INPUT statements).

GOSUB line number

Abbreviation: GOS.

GOSUB *line number* is a program-control statement that directs the execution of a program to a subroutine that begins a *line number*. Unless the subroutine ends the entire program with an END statement, it must conclude with a RETURN statement that will return program operations to the statement that follows GOSUB.

Because GOSUB refers to a specific *line number*, *line number* must be a positive integer constant or numeric expression that refers to a line number in the existing program.

A conditional POP statement that is included within a subroutine makes it possible to nullify the normal, automatic-return feature of a RETURN statement. You can, for instance, execute a GOSUB statement, run a portion of the routine and, if desired, break completely away from the subroutine. Suppose that this sort of conditional statement appears at line 1020 in a subroutine:

1020 IF X > 5 THEN POP:GOTO 100

If variable X happens to take on a value that is greater than 5, the computer will “forget” that it was executing a subroutine, and go immediately to line 100. Some sources regard this optional POP feature as a technique for dynamically transforming a GOSUB operation into a GOTO operation.

GOTO line number

Abbreviation: G.

GOTO *line number* is a program-control statement that directs the execution of a program directly to *line number*. Because *line number* represents a specific line number in the program, it must be a positive integer constant or numeric expression.

Examples:

10 GOTO 100

That will direct the program to line 100.

10 GOTO 10*N

That will direct the program to a line number that is equal to 10 times the numeric value currently assigned to variable N. The expression, of course, must point to a line number that exists in the program at the time.

GRAPHICS *x*

Abbreviation: GR.

The GRAPHICS command sets the ATARI BASIC graphics mode, where *x* is a numeric constant or expression that represents a valid graphics-mode number.

If *x* is a positive integer value between 0 and 8, the command will set the normal graphics modes, clear the screen and establish the normal text windows.

Adding a constant 16 to the value sets the graphics modes 0 through 8, clears the screen, but does not include the normal split-screen text windows.

Adding a constant 32 to the value retains the split-screen text windows, but prevents the mode operation from clearing the screen.

Adding a constant 40 suppresses the text windows and the screen-clearing operations.

See Chapter 4 for further examples and details.

IF *expr* THEN *statement*:*statement* . . .

IF . . . THEN is BASIC's primary conditional statement. Literally it says: IF some arithmetic or logical expression, *expr* is true, THEN execute the specified *statement* or sequence of statements. By implication, if the *statement* is NOT true, then ignore any following *statements*, and GOTO the beginning of the next program line.

Example:

```
IF K$="Y" THEN PRINT "DONE":END
```

That statement literally says: IF the string currently assigned to variable K\$ is a Y, then print DONE and END the program (otherwise, go to the next line of programming).

If the *statement* portion of an IF . . . THEN statement happens to be a GOTO *line number* statement, you can shorten the programming a bit:

IF A <=10 THEN GOTO 100

is the same as

IF A <=10 THEN 100

A line number following the THEN portion of an IF ... THEN statement implies a GOTO operation.

INPUT var1, var2, ...

INPUT #chan,var,var ...

Abbreviation: I.

The simplest form of the INPUT statement halts the execution of a program, prints a question mark on the screen, and awaits a response from the keyboard that ends with a **RETURN** keystroke. The statement assigns the keyboard entry to the var.

Examples:

INPUT X

The program in that instance is expecting a numeric value from the keyboard and will subsequently assign it to variable X.

INPUT X:N(1)=X

INPUT statements cannot directly assign values to subscripted numeric values. The example shows how the INPUT value is first assigned to variable X, and then an assignment statement assigns X to subscripted variable N(1).

INPUT M\$

The program in that instance is expecting a string entry from the keyboard, and it will subsequently assign it to string variable M\$. The variable, of course, must be properly dimensioned at some earlier point in the program.

It is possible to enter more than one numeric or string constant from the keyboard under a single INPUT statement. Such compound INPUT statements must show the variable names separated by commas.

Examples:

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

The INPUT statement is expecting three consecutive numerical constants from the keyboard. The operator has the option of entering all three constants, and separating them with commas before striking the **RETURN** key; or striking the **RETURN** key after each entry.

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

That INPUT statement expects a series of three string-constant entries from the keyboard. Each string entry must conclude with a **RETURN** keystroke.

```
INPUT X,A$
```

The example shows that it is possible to mix variable types within a compound INPUT statement. This one is expecting a numeric constant followed by a string constant.

Unless specified otherwise, it is assumed that the INPUT statement is expecting numeric or string constants from the keyboard—actually, the screen editor assigned to IOCB 0. It is altogether possible, however, to assign the INPUT statement to other input sources.

The general form of the INPUT statement in that case is:

```
INPUT #chan,var
```

where *chan* is the IOCB channel number that is to supply the constant for variable *var*.

The simple INPUT statement cited earlier will always print a prompting question mark. That feature is usually an asset, but it is often inappropriate. The following example allows the operator to enter a numeric constant from the keyboard by first assigning the keyboard to IOCB 1. The question mark will not appear in that instance. (Unfortunately, the entry isn't printed to the screen while it is being entered, either.)


```
10 OPEN #1,4,0,"K:"  
20 PRINT "ENTER A NUMBER BETWEEN 0 AND 9 ";  
30 INPUT #1,X  
40 CLOSE #1
```

See Chapter 6 for additional information regarding the INPUT statement as a dynamic I/O command.

INT(x)

INT(x) is a numeric function that returns the next-smaller integer value of x; where x is any floating-point constant or expression.

Examples:

```
PRINT INT(3.8)
```

That statement will print 3 to the screen.

```
PRINT INT(8.3)
```

That will print 8 to the screen.

```
PRINT INT(-1.2)
```

Bearing in mind that negative values that are farther from zero are considered to be smaller than those closer to zero, that statement will print -2 to the screen.

LEN (x\$)

LEN (x\$) is a string function that returns the number of characters assigned to string x\$.

LET var=expr

Abbreviation: LE.

LET is a variable-assignment statement that assigns a numeric or string constant or expression, *expr*, to a designated numeric or string variable name, *var*.

The variable types must match: numeric values and expressions must be assigned to numeric variable names, and string values and expressions must be assigned to string variable names.

The LET expression is actually optional; *var = expr* is adequate.

Examples:

```
LET X=1+Y
LET X$="FLIP YOUR WIG"
G=X*Y
```

LIST *output device start line, last line*

Abbreviation: L.

The LIST command is used for writing information, or selected portions of it, to the designated *output device* in an ATASCII-coded format. If no *output device* is designated in the command, the system automatically assumes it to be the video display. The terms *first line* and *last line* represent program line numbers that may be used in the following combinations:

LIST—list the entire program, from the lowest-numbered line through the highest-numbered line.

LIST *start line*—list only line number *start line* in the program.

LIST *start line, last line*—list the program, beginning from line number *start line* and ending with line number *last line*.

Values *start line* and *last line* need not refer to specific line numbers in the program, but they must be positive integer values.

Examples:

```
LIST 10
```

That command lists program line 10 to the screen.

```
LIST "P:"
```

That command lists the entire program to the ATARI printer.

LIST "D:TRY.BAS"

That command saves the entire BASIC program as TRY.BAS in ATASCII-coded (nontokenized) format to the default disk drive. See Chapter 6 for further details concerning ATASCII-coded programs and files.

A LISTing operation can be aborted by striking the BREAK or SYSTEM RESET keys, but the user should be aware that such an interruption will render LISTs to the program recorder or disk system useless.

LOAD *input device*

Abbreviation: LO.

LOAD is used for loading BASIC programs or files into the system from the designated *input device*. The specified device must be one that is appropriate for input operations and the programming must have been saved in a tokenized format—by means of a SAVE command. Because programs that are saved by CSAVE or LIST commands are saved in an ATASCII-coded format, they cannot be loaded by means of the LOAD command.

See Chapter 1 for applications of the LOAD command and Chapters 6 and 7 for further information about tokenized BASIC.

LOCATE *col,row,x*

Abbreviation: LOC.

LOCATE is a graphics function that determines the character or graphics code for a specified screen position, and assigns the value to a numerical variable, *x*. The screen position is given in column/row coordinates, *col* and *row*, that are appropriate for the current graphics mode.

In graphics Modes 0, LOCATE returns the ATASCII code number for the character at *col,row*.

In graphics Modes 1 and 2, LOCATE returns a value that indicates both the character being displayed and the color register it is using.

In graphics Modes 3 through 8, LOCATE returns the color register being used for plotting to *col,row*.

See Chapter 4 for further details concerning the formatting of text and graphics data.

The LOCATE statement is not valid until the program executes a GRAPHICS statement, thereby opening IOCB Channel #6 for input from the screen.

Executing the LOCATE statement automatically leaves the print cursor (whether it is actually visible or not) at the column location that immediately follows the *col,row* designation.

LOG(x)

LOG(x) is a numeric function that returns the natural, base-e, logarithm of x; where x is a floating-point constant or numeric expression that is greater than zero.

Use the CLOG(x) function when it is desirable to obtain the base-10 logarithm of x.

LPRINT *expr list*

Abbreviation: LP.

LPRINT is a statement that is quite similar to PRINT, but prints data to an ATARI printer instead of the video screen. It can print lists, numeric constants and expressions, as well as string constants and expressions.

If the items in the *expr list* are separated by a semicolon, the printing of one item begins immediately after the end of a previous one. If the items are separated by a comma, each is printed at the beginning of the next-available tab-stop column on the screen.

LPRINT uses IOCB Channel #7.

NEW

NEW is a command that, in effect, clears all existing BASIC programming from RAM. It also closes all IOCB channels except #0 (which is used for the screen editor), and sets all trigonometric functions to work in units of radians.

It is advisable to execute the NEW command prior to entering a new program.

NEXT

Abbreviation: N.

The NEXT statement is meaningless without a preceding FOR statement. See FOR.

NOT

NOT is a logical operator that is used to negate a logical expression. It is generally used in conditional statements such as:

IF NOT *expression* THEN *statement*

See Chapter 2 for further details.

NOTE #*chan*,*sec*,*byte*

Abbreviation: NO.

The NOTE function returns the sector and byte values at the current disk-file pointer position for a disk device at IOCB channel, *chan*. Those values are assigned to the designated numeric variable, *sec* and *byte*.

The NOTE function is not available under ATARI DOS 1.0.

ON x GOSUB *line1*,*line2* . . .

ON . . . GOSUB is a control statement that calls a subroutine beginning at one of a list of line numbers, *line1*, *line2*, . . . and so on. The line number that is selected is determined by the current value of the integer expression, *x*. If *x* happens to have a value of 1, the statement will call the subroutine specified by the first *line* designation; if *x* is equal to 2, the statement will call the second-specified *line*; if *x* is equal to 8, the statement will call a subroutine that begins at the eighth line number in the list.

If it happens that the current value of x is 0 or if its value exceeds the number of lines designated in the program, the computer ignores the `ON . . . GOSUB` altogether and goes to the next statement.

Examples:

```
10 FOR N=1 TO 5
20 ON N GOSUB 100,110,135,100,150
30 NEXT N
```

As the value of N increments from 1 through 5, line 20 calls subroutines that begin at lines 100, 110, 135, 100 and 150 in that order. (Notice that it is possible to call the same subroutine more than one time within the list.)

```
100 ON X=GOSUB 1000, X*100, 200
```

If X is equal to 0, the computer will ignore this statement and resume execution from the next-available program statement. If X is equal to 1, the statement will send operations to a subroutine beginning at line 1000; and when X is equal to 2, the computer will consider the $X*100$ listing and send the operations to line 200. When X is equal to 3, the program will look at the third line listing, and send operations to a subroutine that begins at line 200. Finally, whenever X is greater than 3, the computer will resume execution from the next-available program statement.

ON x GOTO *line1,line2* . . .

`ON . . . GOTO` is a control statement that selects one of any number of *line* numbers that are listed in the statement. The general features and purposes are nearly identical to those for the `ON . . . GOSUB` statement. The only differences are those that distinguish the simpler `GOTO` and `GOSUB` statements.

OPEN #*chan,task,aux,"dev"*

Abbreviation: O.

The OPEN command represents the most direct and useful means for opening an IOCB channel for service. Some ATARI BASIC functions automatically open certain channels, but a good many operations call for opening and configuring a channel for a particular purpose.

The command has four different arguments. The first, *chan*, designates the channel that is to be opened—1 through 7; and the last, *dev*, designates the type of device the channel is to service. Parameters *task* and *aux* define the operations.

Chapter 6 includes tables for defining those arguments for given devices.

Examples:

```
OPEN #1,4,0,"K:"
```

That opens IOCB Channel #1 for input from the keyboard. The function is especially useful, when in conjunction with a subsequent INPUT #1,X statement, for picking up single-keystroke character codes from the keyboard.

```
OPEN #2,13,0,"R1:"
```

That opens IOCB Channel #1 for read/write operations from RS-232 serial port number 1. The device handler routine, AUTORUN.SYS, must be resident in the system before any serial operation can be performed.

All channels thus opened will be automatically closed when the program reaches its end. You can close all channels during the execution of a program (except #0 for the screen editor) by executing a CLOSE command.

OR

OR is used as a logical operator in conditional statements such as:

```
IF expr1 OR expr2 THEN statement
```

See Chapter 2 for further details.

PADDLE(x)

PADDLE returns an integer value between 1 and 228 that indicates the rotated position of the paddle connected to paddle input *x*. The lowest values indicate a full clockwise turn, and the higher values indicate full counterclockwise turn. There is no meaningful relationship between the values generated by the PADDLE function and the amount of angular rotation.

Example:

```
FOR D=0 TO 100+10*PADDLE(0):NEXT D
```

That example executes a time-delay interval that becomes shorter as paddle 0 is rotated in a clockwise direction. The function in this case could be controlling the speed of a missile figure on the screen.

PEEK(addr)

PEEK is a special-purpose function that returns the decimal value of the byte that is stored at memory address *addr*.

Example:

```
PRINT PEEK(88)+256*PEEK(89)
```

That statement PEEKs into RAM addresses 88 and 89, and prints a decimal value that happens to indicate the starting address of the current screen RAM.

PLOT col,row

Abbreviation: PL.

The PLOT command sets up a graphics operation to plot a character or pixel at the column/row location indicated by *col,row*. Appropriate ranges for numeric expressions *col* and *row* depend on the current graphics screen mode. See Chapter 4 for further details.

POINT #*chan*,*sect*,*byte*

Abbreviation: P.

The POINT statement adjusts the position of the disk-file pointer for the specified channel, *chan*. The adjustment is to sector *sect* and byte *byte*.

The complement of this statement is NOTE. Neither is available under ATARI DOS 1.0.

POKE *addr*,*data*

POKE is a special statement that sends a byte of *data* to the designated memory-mapped address, *addr*. The *data* byte must be a decimal integer value between 0 and 255, and *addr* may be a decimal address between 0 and 65535. It is the complement of the PEEK function.

However, POKEing data to ROM locations, control blocks that are opened for input operations or to RAM addresses where no physical RAM exists has no effect on the system.

The user should avoid careless or indiscriminate use of the POKE command, because POKEing into RAM addresses that have functions already assigned to them by the operating system, BASIC or DOS, can cause a system software "crash."

POP

POP is a special control command that, in effect, forces the computer to abort the automatic return or looping features of GOSUB, ON . . . GOSUB and FOR . . . NEXT operations that are in effect at the time. It actually POPs the return address off the top of the run-time stack.

Practical applications of POP generally concern techniques for breaking out of a subroutine or FOR . . . NEXT loop under a prescribed condition. A general statement imbedded within a subroutine or FOR . . . NEXT loop might look like this:

IF *condition* THEN POP:GOTO *line number*

If the designated *condition* is satisfied, the POP statement effectively clears the way for doing a GOTO to some program *line number* that is outside the subroutine or loop. The program will then progress as though the GOSUB or FOR . . . NEXT loops were never initiated.

Use POP statements sparingly and with care, especially in programs that use extensively nested GOSUBs and FOR . . . NEXT loops.

POSITION *col,row*

Abbreviation: POS.

The POSITION statement moves the print cursor directly to the column/row screen position as designated by the *col* and *row* values or expressions. The range of values that are available for this statement varies with the screen mode. See Chapter 4 for examples and further details.

PRINT *expr list*

Abbreviations: PR. or ?

PRINT is a statement that is used for printing information to the video display. It can print lists of numeric constants and expressions as well as string constants and expressions.

If the items in the *expr list* are separated by a semicolon, the printing of one item begins immediately after the end of a previous one. If the items are separated by a comma, each is printed at the beginning of the next-available tab-stop column on the screen.

The following version of the print statement transfers numeric or string data to an IOCB channel:

```
PRINT #chan; . . .
```

See Chapter 6 for details.

PTRIG *x*

PTRIG returns a value of 0 or 1, depending on whether the pushbutton on paddle device *x* is pressed or not pressed.

Example:

```
IF PTRIG 1=0 THEN END
```

That short routine will end the program if the player happens to be pressing the pushbutton on paddle device 1 when the conditional statement is executed.

PUT #*chan*,*x*

Abbreviation: PU.

The PUT statement outputs a single integer value or expression, *x*, to the designated IOCB channel, *chan*. It is the functional complement of the GET statement.

See Chapter 6 for examples and further details.

RAD

RAD is a statement or command that converts all subsequent expressions of angles to units of radians. Executing a RUN command automatically sets the RAD mode, and you must include a DEG statement into the program whenever you want to work with the degrees format.

READ *var1*, *var2* . . .

Abbreviation: REA.

READ statements must be used in conjunction with DATA statements. The statement sequentially reads items in DATA lines and assigns them to the specified string or numeric variables. The variable types must match; that is, a READ A\$ statement expects a string value from the DATA list, and a READ X statement expects a numeric value. Compounded READ statements may mix variable types as long as the corresponding sequence of DATA items follows the same format.

For example, this combination of READ/DATA statements will work:

```
10 DIM A$(4)
20 READ A$,X
30 DATA BOY,10
```

but this one will not, because it attempts to assign a string constant, GIRL, to a numeric variable, X:

```
10 DIM A$(4)
20 READ A$,X
30 DATA BOY,GIRL
```

It might appear that the next example would return a type-mismatch error, but it does not. The numbers assigned to A\$ and B\$, however, will be regarded as string constants.

```
10 DIM A$(4),B$(4)
20 READ A$,B$
30 DATA 5,6
```

Remember that numbers can be regarded as either numeric or string constants, while string values can be regarded only as strings.

REM

Abbreviations: R. or a period followed by a space

REM is a nonexecutable statement that allows you to insert printed text, or REMarks, within a program listing.

Examples:

```
10 REM ** COUNT TO TEN **
100 R. NOTE: BEGIN PRINTER ROUTINE HERE <<<
```

RESTORE *line number*

Abbreviation: RES.

RESTORE is a program command that allows a DATA list to be READ, beginning from the DATA line having the designated *line number*. If *line number* is omitted from the command, the next READ operation will begin from the lowest-numbered DATA line.

RETURN

Abbreviation: RET.

RETURN is a program-control command that returns operations from a subroutine to the statement following the corresponding GOSUB. The command is meaningless without a calling GOSUB or ON . . . GOSUB statement.

RND(x)

RND is a numeric function that returns a random-number value that is equal to or greater than 0, but less than 1. The *x* term is a dummy argument that has no effect on the function, so it can have any numeric value.

Programs that use randomly generated numbers most often require integer values and, even more often, values that are larger than 1. The trick in such instances is to use a function of this general form:

$$\text{INT}(\text{modulus} * \text{RND}(0)) + \text{lownum}$$

where *modulus* is the number of integers to be included in the random series, and *lownum* is the lowest-valued integer in the series. So if you want to generate random integers between 0 and 9, inclusively, *modulus* = 10 (there are 10 different integers in the range of 0 through 9) and *lownum* = 0. The following program used that format to print a series of 100 random integers separated by spaces:

```
10 FOR N=1 TO 100
20 PRINT INT(10*RND(0)); CHR$(32);
30 NEXT N
```

Or perhaps you need random integers between 50 and 100, inclusively. That represents a *modulus* of 51 and a *lownum* of 50. To see a hundred such numbers:

```
10 FOR N=1 TO 100
20 PRINT INT(51*RND(0))+50; CHR$(32);
30 NEXT N
```

RUN [*input device*]

Abbreviation: RU.

Used without the optional *input device* specification, RUN is a command that initiates the execution of the resident BASIC program from the lowest-numbered line. It closes all I/O control channels and sets the trigonometric-angle format to RAD (radians).

Alternatively, RUN *input device* erases any resident BASIC programming, replaces it with a tokenized BASIC program from the specified *input device*, and begins executing the program from the lowest-numbered line.

Examples:

RUN

That command initiates execution of the resident BASIC program from the lowest-numbered line.

RUN "D:FIXER"

That version loads a tokenized BASIC program named FIXER from the default disk drive, and initiates execution from the lowest-numbered line.

Use GOTO *line number* to begin execution of a resident program from some *line number* other than the lowest-numbered line.

SAVE *output device*

Abbreviation: S.

SAVE is a command that transfers tokenized BASIC programming from the computer to the specified *output device*.

See Chapter 1 for examples and further details.

SETCOLOR *reg, hue, lum*

Abbreviation: SE.

SETCOLOR sets color register *reg* to the specified hue and luminance values, *hue* and *lum*.

The value assigned to *reg* must specify one of the five color registers: 0,1,2,3, or 4.

The value assigned to *hue* ought to be a positive integer between 0 and 15. The colors vary in brightness, or shade, according to the subsequent *lum* value, so the following list of *hue* values and colors is determined with the assumption that the *lum* value is 0.

It is possible to specify a *lum* value as any positive integer. The lowest-valued *lum* factors that cover the entire luminance range are even numbers between 0 and 14, inclusively; where 0 sets up the darkest shade and 14 sets up the lightest.

See Chapter 4 for tables of hue and luminance values.

SGN(*x*)

SGN(*x*) is a numeric function that returns integer values of -1, 0, or 1, depending on the sign of *x*; where *x* is any floating-point constant or numeric expression.

If *x* is negative, SGN(*x*) returns a value of -1

If *x* is equal to 0, SGN(*x*) returns a value of 0

If *x* is positive, SGN(*x*) returns a value of 1

SIN(*x*)

SIN(*x*) is a numeric function that returns the trigonometric sign value of angle *x*; where *x* is any floating-point constant or numeric expression. The angle is normally assumed to be expressed in radian measure, but it can be expressed in degrees by preceding the function with a DEG command.

SOUND *voice, pitch, dist, vol*

Abbreviation: SO.

The SOUND statement causes a tailored sound to be produced at the loudspeaker in the tv receiver or monitor. The four arguments *voice*, *pitch*, *dist*, and *vol* determine the voice, pitch, amount of distortion, and volume, respectively.

There are four voices that are numbered 0 through 3. It is possible to select any combination of them. The pitch is set by assigning an integer value between 0 and 255, where 0 is the highest available pitch and 255 is the lowest. See Chapter 7 for a table of pitch values and corresponding musical notes.

The distortion figure determines the amount of noise, or tonal distortion, that is injected into the sound. The values must be even-numbered integers, where the lower values produce more distortion than the higher values do.

The volume of the sound is set by assigning integer values between 0 and 15, where a value of 0 completely silences the voice, and 15 produces the loudest sound.

Once a SOUND command is executed, it generates its sound until that voice is told to do otherwise, or you execute certain commands that affect the normal operation of BASIC programs.

See Chapter 7 for examples and further details.

SQR(x)

SQR(x) is a numeric function that returns the square-root of x ; where x is any positive-valued, floating-point constant or numeric expression.

STATUS #*chan*, *numvar*

Abbreviation: ST.

STATUS assigns the status code of the last-used IOCB channel, *chan*, to a numerical variable, *numvar*. Table 3-2 lists the status codes and their interpretations.

Table 3-2. Status Codes That Are Returned by the STATUS Command

Code	Meaning
001	Operation complete (no errors)
003	End of file (EOF)
128	BREAK executed
129	IOCB channel already in use (OPEN)
130	Nonexistent device
131	Opened for WRITE only
132	Invalid command
133	Device or file not open
134	Invalid IOCB channel number
135	Opened for READ only
136	End of file (EOF) encountered
137	Truncated record
138	Device timeout (doesn't respond)
139	Device NAK
140	Serial bus input framing error
141	Cursor out of range
142	Serial bus data frame overrun error
143	Serial bus data frame checksum error
144	Device-done error
145	Bad screen mode
146	Function not supported by handler
147	Insufficient memory for screen mode
160	Disk drive number error
161	Too many open disk files
162	Disk full
163	Fatal disk I/O error
164	Internal file number mismatch
165	Filename error
166	Point data length error
167	File locked
168	Command invalid for disk
169	Disk directory full
170	File not found
171	Point invalid

STEP

STEP is an optional part of a FOR . . . NEXT statement, and thus does not function alone. See FOR.

STICK(*jsno*)

STICK returns one of eight different integer values that indicate the position of the joystick handle connected to joystick input number *jsno*. Fig. 3-1 shows the joystick positions and their values.

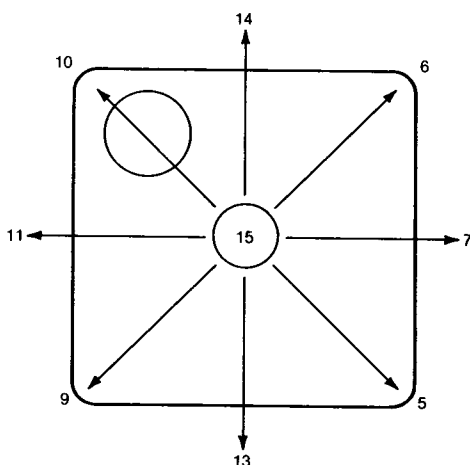


Fig. 3-1. Joystick positions and the values returned by the joystick statements.

The relationships between *jsno* and the game-controller connections are:

Controller	<i>jsno</i>
1	0
2	1
3	2
4	3

Example:

IF STICK(0)=14 THEN DY=DY+1

That conditional statement will increment the value of DY if the joystick that is connected to game controller #1 is pushed straight up.

STRIG(*jsno*)

STRIG refers to the condition of the pushbutton on the joystick that is connected to game controller number *jsno*. The statement returns a value of 0 whenever the pushbutton is depressed, and a value of 1 whenever it is not depressed. (See a table of game-controller input numbers and *jsno* under STICK.)

STOP

Abbreviation: STO.

The STOP command halts the execution of a program and causes the computer to return a message, STOPPED AT LINE _____. STOP does not close any files, so the program can be resumed from the STOP command by entering the CONT command.

STOP is often inserted temporarily into a program to halt program operations for debugging purposes. It is rarely included in programs that are intended for use by anyone but the programmer.

STR\$(*x*)

STR\$ converts any valid, floating-point numeric value, *x*, to a string equivalent.

See Chapter 2 for a detailed discussion of numeric and string values.

THEN

THEN is an integral part of IF . . . THEN conditional statements, and makes no sense without the IF statement. See IF.

TO

TO is an integral part of a FOR . . . NEXT statement, and thus makes no sense outside that context. See FOR.

TRAP *lineno*

Abbreviation: T.

The TRAP statement is used for dealing with statements that might otherwise cause annoying error-message interrupts. A classic instance appears in the following routine:

```
10 PRINT "ENTER A NUMBER"  
20 INPUT X  
30 PRINT 1/X  
40 GOTO 10
```

The idea is simple: prompt the user to enter a number, then display its inverse—that number divided into 1. Everything works fine until the user enters a 0. That brings up a divide-by-0 error signal.

The following modification uses a TRAP statement to deal with that situation in a far more graceful fashion:

```
10 PRINT "ENTER A NUMBER"  
20 INPUT X  
25 TRAP 100  
30 PRINT 1/X  
40 GOTO 10  
100 PRINT "INFINITY"  
120 GOTO 10
```

A TRAP statement must always precede the statement that contains the potential error situation—at line 25 in this instance. The *lineno* portion of the TRAP statement directs program operations to that line, but only in the event of an error condition. This program will “trap” to line 100 only if the operator happens to enter a 0 in response to the INPUT statement in line 20.

A truly professional piece of BASIC programming will use TRAP statements to avoid error problems whenever users are not running the system properly. Commercial software should never “crash” because of an error-producing situation that can be dealt with more gracefully with TRAP routines.

USR(addr)

The USR function directs the computer out of BASIC and to a machine-language routine that begins at address *addr*. Assuming that the machine-language programming is properly structured, the computer will execute the program and return to BASIC at the statement following the USR.

It is possible to carry any number of numeric values, or 2-byte memory addresses that point to numeric or string values, by adding appropriate arguments to the USR function. Those arguments are placed, *in right-to-left order*, on the system's hardware stack. Consider this example:

USR(7148,128,ADR(M\$))

This USR function calls a machine-language subroutine that begins at decimal memory address 7148. It first pushes the line number for the next BASIC command onto the hardware stack, followed by the value returned by the ADR(M\$) function (the memory address of the content of string variable M\$), a 2-byte version of constant 128 and, finally, a 1-byte integer indicating the number of arguments in the function. The actual machine-language routine is then run.

An assembly language RET returns operations to the BASIC instruction line that was originally pushed onto the stack. The assumption, however, is that the instruction line number is now residing on the top of the stack; and that suggests that all other data, including that pushed to the stack by the USR function, itself, must be pulled off before executing the RET instruction.

See Chapter 7 for examples and further details.

VAL(x\$)

VAL converts numerals that are represented in a string format, x\$, into their corresponding numeric value. It is the inverse of the STR\$(x) function.

XIO cmd, #chan, param1, param2, dev

Abbreviation: X.

The XIO command might well be the most versatile I/O command in ATARI BASIC. It can direct the flow of data into and out of any available IOCB channel, work with any sort of computer-compatible device, format special serial operations, and even fill in blocks of graphics on the screen.

See Chapter 6 for a special treatment of this command, including tables that define its parameters.

Chapter 4

The Text and Graphics Screens

The ATARI system features 8 different screen modes and 21 meaningful variations of them. Table 4-1 summarizes those screen formats.

Modes 0, 1, and 2 are usually classified as text and character modes, primarily because they are used for printing characters from a well-defined, bit-mapped character set. Modes 3 through 8, and their variations, are regarded as graphics modes because they plot pixels of color to the screen. The graphics modes are further divided into 4-color, 2-color, and 1-color modes. The discussions in this chapter are organized according to these classifications.

FEATURES COMMON TO ALL SCREEN MODES

Hue and luminance values, and the RAM addresses of the five color registers, are common to all ATARI screen modes. Table 4-2 lists the hue values, Table 4-3 shows the luminance values, and Fig. 4-1 shows the basic arrangement of the color registers and their RAM addresses.

NOTE: Color is defined throughout this book as the combination of hue and luminance values. There are 16 possible hue values and 8 luminance values; and that implies that ATARI graphics features 128 basic colors.

**Table 4-1. Summary of ATARI
Home Computer Screen Modes**

Graphics Mode	Size (col × row)	Text Window (col × row)	Notes
0	40 × 24	—	Text only
1	20 × 20	40 × 4	Expanded characters with text window
2	20 × 10	40 × 4	Expanded characters with text window
3	40 × 20	40 × 4	Low-resolution, 4-color graphics with text window
4	80 × 40	40 × 4	Medium-resolution, 2-color graphics with text window
5	80 × 40	40 × 4	Medium-resolution, 4-color graphics with text window
6	160 × 80	40 × 4	High-resolution, 2-color graphics with text window
7	160 × 80	40 × 4	High-resolution, 4-color graphics with text window
8	320 × 160	40 × 4	Very high resolution, 2-color graphics with text window
17	20 × 24	none	Same as Mode 1 without text window
18	20 × 12	none	Same as Mode 2 without text window
19	40 × 24	none	Same as Mode 3 without text window
20	80 × 48	none	Same as Mode 4 without text window
21	80 × 48	none	Same as Mode 5 without text window
22	160 × 96	none	Same as Mode 6 without text window
23	160 × 96	none	Same as Mode 7 without text window
24	320 × 192	none	Same as Mode 8 without text window

**Table 4-1—cont. Summary of ATARI
Home Computer Screen Modes**

Graphics Mode	Size (col × row)	Text Window (col × row)	Notes
32	40 × 24	—	Same as Mode 0 without screen clearing
33	20 × 20	40 × 4	Same as Mode 1 without screen clearing
34	20 × 10	40 × 4	Same as Mode 2 without screen clearing
35	40 × 20	40 × 4	Same as Mode 3 without screen clearing
36	80 × 40	40 × 4	Same as Mode 4 without screen clearing
37	80 × 40	40 × 4	Same as Mode 5 without screen clearing
38	160 × 80	40 × 4	Same as Mode 6 without screen clearing
39	160 × 80	40 × 4	Same as Mode 7 without screen clearing
40	320 × 160	40 × 4	Same as Mode 8 without screen clearing
49	20 × 24	none	Same as Mode 17 without screen clearing
50	20 × 12	none	Same as Mode 18 without screen clearing
51	40 × 24	none	Same as Mode 19 without screen clearing
52	80 × 48	none	Same as Mode 20 without screen clearing
53	80 × 48	none	Same as Mode 21 without screen clearing
54	160 × 96	none	Same as Mode 22 without screen clearing
55	160 × 96	none	Same as Mode 23 without screen clearing
56	320 × 192	none	Same as Mode 24 without screen clearing

Table 4-2. Standard ATARI Home Computer Hue Values and Those Values as Multiplied by 16

Color	Hue Value	16*Hue Value
Black	0	0
Brown	1	16
Red-orange	2	32
Dark orange	3	48
Red	4	64
Deep lavender	5	80
Deep blue-green	6	96
Ultramarine blue	7	112
Medium blue	8	128
Deep blue	9	144
Blue-gray	10	160
Olive	11	176
Medium green	12	192
Deep green	13	208
Orange-green	14	224
Orange	15	240

NOTE: All color values assume a luminance of 0. Their actual appearance depends on the color adjustments on the tv or monitor, and the names, themselves, are subject to personal interpretation.

Table 4-3. Relevant Luminance Values

Relative Luminance	Value
Darkest	0
	2
	4
	6
	8
	10
	12
Lightest	14

NOTE: Odd-numbered integers in the range of 1 through 15 produce the same luminance as the preceding even-numbered value.

Fig. 4-1. The five standard color registers and their decimal RAM addresses.

REGISTER 0 ADDRESS 708
REGISTER 1 ADDRESS 709
REGISTER 2 ADDRESS 710
REGISTER 3 ADDRESS 711
REGISTER 4 ADDRESS 712

The ATARI system uses a ROM-based character set that is used for plotting text characters and a handful of special graphics characters. Each character has a code number assigned to it, but there are two different coding schemes.

One coding scheme, shown in Table 4-4, uses code numbers that are closely related to the customary ASCII format (the addition of some special graphics characters prompts most writers to refer to it as the ATARI ASCII, or ATASCII, character set). These are the character codes that are relevant to BASIC functions such as `CHR$` and `ASC`.

Table 4-5 shows the ATARI *internal* character set and corresponding codes. These are the values that are actually carried as data in the screen RAM address locations; and from a programmer's point of view, they are the values that are most relevant when `POKE`ing and `PEEK`ing characters into the screen RAM area. The table does not include the inverse characters, but they are plotted by setting the most-significant bit of the character data byte to 1, or adding 128 to the code numbers shown here.

Generally speaking, the print cursor is a mechanism that points to a position on the screen when the next character or pixel of color is to be plotted. All screen modes share several sets of registers that are directly related to the position of the print cursor, and the addresses and functions are summarized for you in Table 4-6.

Table 4-4. The ATASCII-Coded Character Set



















Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
0		NUL	CTRL-,
1		SOH	CTRL-A
2		STX	CTRL-B
3		ETX	CTRL-C
4		EOT	CTRL-D
5		ENQ	CTRL-E
6		ACK	CTRL-F
7		BEL	CTRL-G
8		BS	CTRL-H
9		HT	CTRL-I
10		LF	CTRL-J
11		VT	CTRL-K
12		FF	CTRL-L
13		CR	CTRL-M
14		SO	CTRL-N
15		SI	CTRL-O
16		DLE	CTRL-P
17		DC1	CTRL-Q

Table 4-4—cont. The ATASCII-Coded Character Set


















Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
18		DC2	CTRL-R
19		DC3	CTRL-S
20		DC4	CTRL-T
21		NAK	CTRL-U
22		SYN	CTRL-V
23		ETB	CTRL-W
24		CAN	CTRL-X
25		EM	CTRL-Y
26		SUB	CTRL-Z
27		ESC	ESC/ESC
28		FS	ESC/CTRL--
29		GS	ESC/CTRL-=
30		RS	ESC/CTRL-+
31		US	ESC/CTRL-*
32		(space)	Space Bar
33		!	SHIFT-1
34		"	SHIFT-2

Table 4-4—cont. The ATASCII-Coded Character Set


















Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
35		#	SHIFT-3
36		\$	SHIFT-4
37		%	SHIFT-5
38		&	SHIFT-6
39		'	SHIFT-7
40		(SHIFT-9
41	)	SHIFT-0
42		*	*
43		+	+
44		,	,
45		-	-
46		.	.
47		/	/
48		0	0
49		1	1
50		2	2
51		3	3

Table 4-4—cont. The ATASCII-Coded Character Set

Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
52	4	4	4
53	5	5	5
54	6	6	6
55	7	7	7
56	8	8	8
57	9	9	9
58	:	:	SHIFT-;
59	;	;	;
60	<	<	<
61	=	=	=
62	>	>	>
63	?	?	SHIFT- /
64	@	@	SHIFT-8
65	A	A	A
66	B	B	B
67	C	C	C
68	D	D	D

Table 4-4—cont. The ATASCII-Coded Character Set

Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
69	E	E	E
70	F	F	F
71	G	G	G
72	H	H	H
73	I	I	I
74	J	J	J
75	K	K	K
76	L	L	L
77	M	M	M
78	N	N	N
79	O	O	O
80	P	P	P
81	Q	Q	Q
82	R	R	R
83	S	S	S
84	T	T	T
85	U	U	U

Table 4-4—cont. The ATASCII-Coded Character Set


















Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
86		V	V
87		W	W
88		X	X
89		Y	Y
90		Z	Z
91		[SHIFT-;
92		\	SHIFT-,
93	]	SHIFT-+
94		^	SHIFT-*
95		_	SHIFT--
96		\	CTRL-
97		a	LOWR A
98		b	LOWR B
99		c	LOWR C
100		d	LOWR D
101		e	LOWR E
102		f	LOWR F

Table 4-4—cont. The ATASCII-Coded Character Set

Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
103	g	g	LOWR G
104	h	h	LOWR H
105	i	i	LOWR I
106	j	j	LOWR J
107	k	k	LOWR K
108	l	l	LOWR L
109	m	m	LOWR M
110	n	n	LOWR N
111	o	o	LOWR O
112	p	p	LOWR P
113	q	q	LOWR Q
114	r	r	LOWR R
115	s	s	LOWR S
116	t	t	LOWR T
117	u	u	LOWR U
118	v	v	LOWR V
119	w	w	LOWR W

Table 4-4—cont. The ATASCII-Coded Character Set



























Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
120		x	LOWR X
121		y	LOWR Y
122		z	LOWR Z
123		{	CTRL-;
124			SHIFT-=
125		}	ESC/CTRL-<
126		~	ESC/BACK S
127		DEL	ESC/TAB
128		NUL	() CTRL-,
129		SOH	() CTRL-A
130		STX	() CTRL-B
131		ETX	() CTRL-C
132		EOT	() CTRL-D
133		ENQ	() CTRL-E
134		ACK	() CTRL-F
135		BEL	() CTRL-G
136		BS	() CTRL-H

Table 4-4—cont. The ATASCII-Coded Character Set


















Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
137		HT	(⌘) CTRL-I
138		LF	(⌘) CTRL-J
139		VT	(⌘) CTRL-K
140		FF	(⌘) CTRL-L
141		CR	(⌘) CTRL-M
142		SO	(⌘) CTRL-N
143		SI	(⌘) CTRL-O
144		DLE	(⌘) CTRL-P
145		DC1	(⌘) CTRL-Q
146		DC2	(⌘) CTRL-R
147		DC3	(⌘) CTRL-S
148		DC4	(⌘) CTRL-T
149		NAK	(⌘) CTRL-U
150		SYN	(⌘) CTRL-V
151		ETB	(⌘) CTRL-W
152		CAN	(⌘) CTRL-X
153		EM	(⌘) CTRL-Y

Table 4-4—cont. The ATASCII-Coded Character Set































Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
154		SUB	() CTRL-Z
155		ESC	() RETURN
156		FS	ESC/SHIFT-BACK S
157		GS	ESC/SHIFT->
158		RS	ESC/CTRL-TAB
159		US	ESC/SHIFT-TAB
160		(space)	() Space Bar
161		!	() SHIFT-1
162		"	() SHIFT-2
163		#	() SHIFT-3
164		\$	() SHIFT-4
165		%	() SHIFT-5
166		&	() SHIFT-6
167		'	() SHIFT-7
168		(() SHIFT-9
169	)	() SHIFT-0
170		*	() *

Table 4-4—cont. The ATASCII-Coded Character Set



































Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
171		+	 +
172		,	 ,
173		—	 —
174		.	 .
175		/	 /
176		0	 0
177		1	 1
178		2	 2
179		3	 3
180		4	 4
181		5	 5
182		6	 6
183		7	 7
184		8	 8
185		9	 9
186		:	 SHIFT-;
187		;	 ;

Table 4-4—cont. The ATASCII-Coded Character Set

Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
188	<	<	(↵) <
189	=	=	(↵) =
190	>	>	(↵) >
191	?	?	(↵) SHIFT-/
192	@	@	(↵) SHIFT-8
193	A	A	(↵) A
194	B	B	(↵) B
195	C	C	(↵) C
196	D	D	(↵) D
197	E	E	(↵) E
198	F	F	(↵) F
199	G	G	(↵) G
200	H	H	(↵) H
201	I	I	(↵) I
202	J	J	(↵) J
203	K	K	(↵) K
204	L	L	(↵) L

Table 4-4—cont. The ATASCII-Coded Character Set



































Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
205		M	 M
206		N	 N
207		O	 O
208		P	 P
209		Q	 Q
210		R	 R
211		S	 S
212		T	 T
213		U	 U
214		V	 V
215		W	 W
216		X	 X
217		Y	 Y
218		Z	 Z
219		[ SHIFT-,
220		\	 SHIFT-+
221	]	 SHIFT-,

Table 4-4—cont. The ATASCII-Coded Character Set



































Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
222		&	() SHIFT-*
223		—	() SHIFT--
224		~	() CTRL-.
225		a	() LOWR A
226		b	() LOWR B
227		c	() LOWR C
228		d	() LOWR D
229		e	() LOWR E
230		f	() LOWR F
231		g	() LOWR G
232		h	() LOWR H
233		i	() LOWR I
234		j	() LOWR J
235		k	() LOWR K
236		l	() LOWR L
237		m	() LOWR M
238		n	() LOWR N

Table 4-4—cont. The ATASCII-Coded Character Set


































Code	ATASCII Character	ASCII Character or Control	Keystroke(s)
239		o	() LOWR O
240		p	() LOWR P
241		q	() LOWR Q
242		r	() LOWR R
243		s	() LOWR S
244		t	() LOWR T
245		u	() LOWR U
246		v	() LOWR V
247		w	() LOWR W
248		x	() LOWR X
249		y	() LOWR Y
250		z	() LOWR Z
251			() CTRL-;
252			() SHIFT-=
253			ESC/CTRL-2
254		~	() ESC/CTRL-BACK S
255		DEL	() ESC/CTRL->

Table 4-5. The ATARI's Internal Character Set

















Code	Character	ROM Addresses	
		Start	End
0		57344	57351
1		57352	57359
2		57360	57367
3		57368	57375
4		57376	57383
5		57384	57391
6		57392	57399
7		57400	57407
8		57408	57415
9		57416	57423
10		57424	57431
11		57432	57439
12		57440	57447
13		57448	57455
14		57456	57463
15		57464	57471

Table 4-5—cont. The ATARI's Internal Character Set

Code	Character	ROM Addresses	
		Start	End
16	0	57472	57479
17	1	57480	57487
18	2	57488	57495
19	3	57496	57503
20	4	57504	57511
21	5	57512	57519
22	6	57520	57527
23	7	57528	57535
24	8	57536	57543
25	9	57544	57551
26	÷	57552	57559
27	×	57560	57567
28	<	57568	57575
29	=	57576	57583
30	>	57584	57591
31	?	57592	57599

Table 4-5—cont. The ATARI's Internal Character Set

Code	Character	ROM Addresses	
		Start	End
32	Q	57600	57607
33	A	57608	57615
34	B	57616	57623
35	C	57624	57631
36	D	57632	57639
37	E	57640	57647
38	F	57648	57655
39	G	57656	57663
40	H	57664	57671
41	I	57672	57679
42	J	57680	57687
43	K	57688	57695
44	L	57696	57703
45	M	57704	57711
46	N	57712	57719
47	O	57720	57727

Table 4-5—cont. The ATARI's Internal Character Set

Code	Character	ROM Addresses	
		Start	End
48	P	57728	57735
49	Q	57736	57743
50	R	57744	57751
51	S	57752	57759
52	T	57760	57767
53	U	57768	57775
54	V	57776	57783
55	W	57784	57791
56	X	57792	57799
57	Y	57800	57807
58	Z	57808	57815
59	[57816	57823
60	\	57824	57831
61]	57832	57839
62	^	57840	57847
63	_	57848	57855

Table 4-5—cont. The ATARI's Internal Character Set

















Code	Character	ROM Addresses	
		Start	End
64		57856	57863
65		57864	57871
66		57872	57879
67		57880	57887
68		57888	57895
69		57896	57903
70		57904	57911
71		57912	57919
72		57920	57927
73		57928	57935
74		57936	57943
75		57944	57951
76		57952	57959
77		57960	57967
78		57968	57975
79		57976	57983

Table 4-5—cont. The ATARI's Internal Character Set

















Code	Character	ROM Addresses	
		Start	End
80		57984	57991
81		57992	57999
82		58000	58007
83		58008	58015
84		59016	58023
85		58024	58031
86		58032	58039
87		58040	58047
88		58048	58055
89		58056	58063
90		58064	58071
91		58072	58079
92		58080	58087
93		58088	58095
94		58096	58103
95		58104	58111

Table 4-5—cont. The ATARI's Internal Character Set

















Code	Character	ROM Addresses	
		Start	End
96		58112	58119
97		58120	58127
98		58128	58135
99		58136	58143
100		58144	58151
101		58152	58159
102		58160	58167
103		58168	58175
104		58176	58183
105		58184	58191
106		58192	58199
107		58200	58207
108		58208	58215
109		58216	58223
110		58224	58231
111		58232	58239

Table 4-5—cont. The ATARI's Internal Character Set

















Code	Character	ROM Addresses	
		Start	End
112		58240	58247
113		58248	58255
114		58256	58263
115		58264	58271
116		58272	58279
117		58280	58287
118		58288	58295
119		58296	58303
120		58304	58311
121		58312	58319
122		58320	58327
123		58328	58335
124		58336	58343
125		58344	58351
126		58352	58359
127		58360	58367

Table 4-6. Cursor-Position Registers

Label	Address	Purpose
ROWCRS	84	Row location of the active cursor
COLCRS	85	LSB of column location of the active cursor
	86	MSB of column location of the active cursor
OLDROW	90	Current row location of the drawing cursor
OLDCOL	91	LSB of current column location of the drawing cursor
	92	MSB of the current column location of the drawing cursor
NEWROW	96	Destination row location for the drawing cursor
NEWCOL	97	LSB of destination column location of the drawing cursor
	98	MSB of destination column location of the drawing cursor
TXTRW	656	Swapped text/drawing row location of the cursor
TXTCOL	657	LSB of swapped text/drawing column location of the cursor
	658	MSB of swapped text/drawing column location of the cursor
CRSNH	752	Cursor inhibit; 0=ON, 1=OFF

Cursor-related BASIC statements make full use of those registers, and it is possible to POKE new cursor positions into them.

MODE-0 GRAPHICS

The Mode-0 screen is the most-used text screen, and it is the system's default screen. Its characteristics apply to the text-window portion of other screen modes that use the split-screen feature.

It is usually adequate to enter this screen mode by doing a **SYSTEM RESET** keystroke, but a few lesser-used text operations are valid only by first executing the GRAPHICS 0 command, or opening IOCB Channel 6 for read/write operations for the screen device.

Organization of the Mode-0 Color Registers

Fig. 4-2 shows the organization of the ATARI's five color registers for the Mode-0 text screen. The same organization, incidentally, applies to the text-window portions of other split-screen graphics modes.

MODE-0 USAGE	
NOT USED	REGISTER 0 ADDRESS 708
CHARACTER LUMINANCE	REGISTER 1 ADDRESS 709
CHARACTER HUE AND BACKGROUND COLOR	REGISTER 2 ADDRESS 710
NOT USED	REGISTER 3 ADDRESS 711
BORDER COLOR	REGISTER 4 ADDRESS 712

Fig. 4-2. Organization of color registers for the Mode-0 text screen.

Color register 1 carries the luminance value for the text characters. The hue value assigned to Register 2 applies to both the background and foreground characters, while the luminance value applies only to the background. It is thus impossible to assign different hue values to the Mode-0 characters and background; the relative luminance values assigned to registers 1 and 2 determine the contrast between the background and characters.

Register 4 carries the full color definition for the Mode-0 border.

From BASIC, the Mode-0 color registers can be set by applying the SETCOLOR statement or by POKEing the appropriate data directly into the registers. The registers can, of course, be directly loaded from a machine-language program as well.

The SETCOLOR statement refers directly to the color-register number, hue values and luminance values:

```
SETCOLOR 1,0,lum  
SETCOLOR 2,hue,lum  
SETCOLOR 4,hue,lum
```

Those three statements, in turn, set the luminance of the foreground characters, the background color and character hue, and the border color.

When using POKE statements to set the color registers, the *addr* portion of the statement refers to the address of the color register and the *data* portion refers to the overall color designation: $16 * \text{hue} + \text{lum}$. So in a general sense, a POKE statement for setting the color registers takes this form:

```
POKE addr,  $16 * \text{hue} + \text{lum}$ 
```

The following POKE statements, in turn, set the luminance of the foreground characters, the background color and character hue, and the border color.

```
POKE 709,lum  
POKE 710, $16 * \text{hue} + \text{lum}$   
POKE 712, $16 * \text{hue} + \text{lum}$ 
```

Working With the Mode-0 Column/Row Format

Fig. 4-3 shows the Mode-0 screen as organized in a column/row format. It shows 40 columns (labeled 0 through 39) and 24 rows (labeled 0 through 23). That is, in a manner of speaking, the equivalent of a horizontal-vertical coordinate system; and it is most convenient when using BASIC statements that refer directly to *col* and *row* parameters and use the print-cursor features.

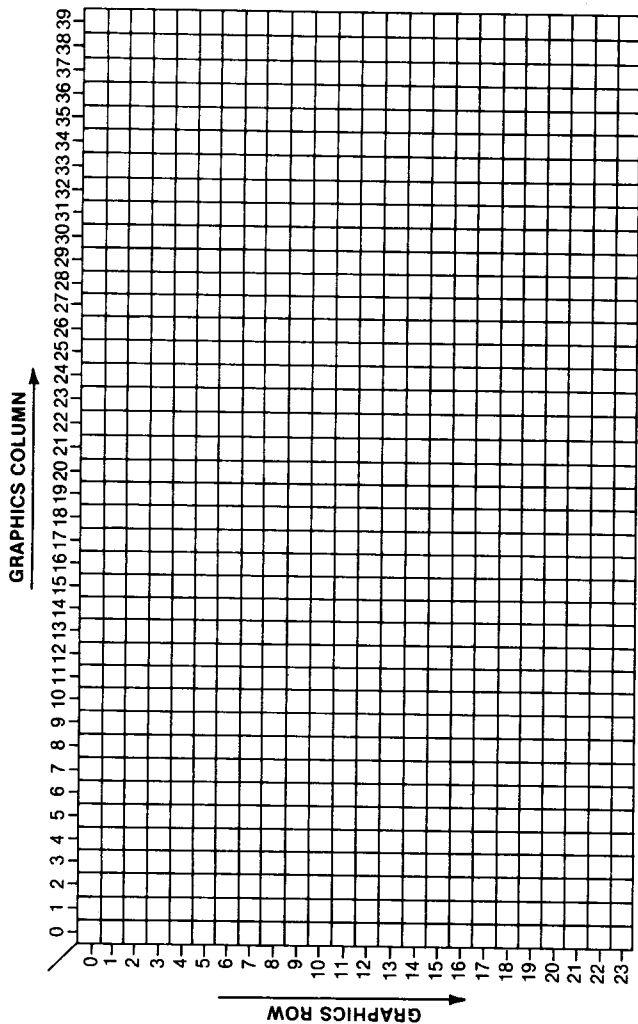


Fig. 4-3. The column/row format for the Mode-0 text screen.

BASIC's PRINT statement provides the most convenient means for printing text information to the Mode-0 screen. Although the print cursor is normally invisible during the execution of a program, every character that is PRINTed to the screen affects the cursor's position. Generally, that means plotting a character to the screen and then moving the cursor to the next-available column location. If a line of text happens to exceed column-location 39, printing will resume from the first column in the next-available row. The print cursor responds to an EOL (end-of-line) character by moving to the beginning of the next-available line on the screen. End-of-screen scrolling is in effect throughout Mode-1 text-printing operations.

In Mode 0, PRINT statements directly affect the cursor locations in ROWCRS and COLCRS (see Table 4-6).

Working With the Mode-0 Margins

Normal Mode-0 PRINT operations do not print text into the first two column locations of each line. The system automatically assigns this 2-column margin at the left side of the screen; and under those circumstances, PRINT operations are confined to columns 2 through 39.

It is possible, however, to adjust those print margins by POKEing some appropriate values into the following RAM locations:

LMARGN 82 Column number of left margin

RMARGN 83 Column number of right margin

The default values for LMARGN and RMARGN are 2 and 39, respectively. You can PRINT into the full row by eliminating the left margin:

POKE 82,0

Other margin settings are possible, but the column value POKED to RMARGN must be greater than the value POKED to LMARGN. Doing a **SYSTEM RESET** restores the default text-margin settings.

Using the POSITION Statement

The `POSITION col,row` statement, when used with the Mode-0 screen, refers directly to the screen's column/row format. In that context, the values assigned to `col` and `row` must be 0 through 39 and 0 through 23, respectively.

`POSITION` is most useful for formatting text on the screen, and the statement makes it possible to see the print cursor outside the current margin boundaries.

`POSITION` directly affects the values that are assigned to the `ROWCRS` and `COLCRS` registers (see Table 4-6); and one can, in fact, mimic the function of a `POSITION` statement by `POKE`ing the same `col` and `row` values into those registers.

Alternative Column/Row Techniques

The matter of plotting text to the Mode-0 screen is not limited to using `PRINT` statements. Three statements that are normally regarded as purely graphics statements can be used with the Mode-0 text screen as well: `COLOR`, `PLOT`, `DRAWTO` and `XIO 18`.

NOTE: The `COLOR`, `PLOT`, `DRAWTO` and `XIO 18` statements function as described in Mode 0 only if IOCB 6 is open; and that is most easily accomplished by preceding the operations with a `GR.0` command.

`PLOT col,row` is normally used for plotting a colored pixel to the designated column/row coordinate of a graphics screen. The color of the pixel in such instances is determined by the most recently executed `COLOR reg` statement, where `reg` refers to the color residing in a particular color register. Used with the Mode-0 screen, however, `PLOT` prints a single character to the designated point on the screen, and the character that is thus printed is determined by the most recent `COLOR` statement, where the `COLOR` parameter is the ATASCII code number (see Table 4-4). Consider the following sequence:

```
GR.0:COLOR 65:PLOT 10,5
```

That will print an upper-case letter A (ATASCII code 65) to column 10, row 5.

And applying a bit of imagination, it is also possible to create some special text effects with the DRAWTO statement. Normally, DRAWTO col,row will draw a line of color from a PLOT point to the designated column/row coordinate. The last-executed COLOR statement determines the color of that line. When working with the Mode-0 screen, however, the parameter assigned to the COLOR statement determines the ATASCII character to be plotted between PLOT-specified place and the DRAWTO-specified place. The following routine plots lower-case letter as from column 0, row 0 to column 39, row 20:

```
GR.0:COLOR 97:PLOT 0,0:DRAWTO 39,20
```

These operations do not affect the text-cursor registers, ROWCRS and COLCRS; rather they work with OLDROW, OLDROW, NEWROW and NEWCOL (see Table 4-6).

The XIO 18 statement can fill a section of the screen with a specified character. For example:

```
10 GRAPHICS 0
20 PLOT 20,10:DRAWTO 20,0:DRAWTO 0,0:POSITION
   0,10
30 POKE 765,97
40 XIO 18,#6,0,0,"S:"
```

Program line 10 sets screen 0 (and opens IOCB #6 for screen operations), and line 20 defines the outline of a rectangular field on the screen. Line 30 POKES the ATASCII code for the desired character, and line 40 executes the "fill" operation.

Using LOCATE, GET, and PUT in Mode-0

LOCATE, GET, and PUT also deal with the Mode-0 screen in a column/row format; and like the graphics statements described in the previous section, IOCB Channel 6 must be open in order to use them.

Executing a PUT #6,x plots the ATASCII character represented by code x to the screen. The position is determined by the main print cursor registers, ROWCRS and COLCRS, and the execution of the statement advances the cursor position. In this context, the PUT statement is easier to use than the COLOR/PLOT combination.

The GET statement is the complement of PUT. Rather than plotting a specified character to the current cursor location, it returns the ATASCII code value of any character that might be printed there. The general form is:

GET #6,numvar

where *numvar* is a numerical variable that takes on the ATASCII value determined by the GET statement.

The LOCATE function is quite similar to GET, but LOCATE does not operate according to the current cursor position. In fact, it operates entirely independent of it. Executing a statement of this general form:

LOCATE col,row,numvar

assigns the ATASCII code number of any character printed to Mode-0 location *col,row* to numerical variable, *numvar*.

The Mode-0 Screen RAM Format

It is quite often helpful to view the Mode-0 screen in terms of the screen RAM addresses and data. Fig. 4-4 shows the addressing ranges for each line on the screen.

This view of the Mode-0 screen facilitates the use of POKE and PEEK statements and machine-language routines for text operations. (The version in the Appendices lists both the decimal and hexadecimal address locations.)

POKE and PEEK statements both refer to an address and a byte of data that is associated with that address. The address in this case refers to the absolute screen address locations shown in the table; the data refers to the internal character codes cited in Table 4-5.

Plotting a character to the screen is thus a matter of executing a statement of this general form:

POKE *addr,data*

where *addr* is the screen address that represents the desired printing location on the screen, and *data* is the internal character code of the character to be printed there. POKEing *data* values larger than 127 will print the inverse version of the internal character set.

In this context, the following sort of PEEK statement returns the internal-code value of the character residing at the specified address:

PEEK(*addr*)

MODE-1 AND MODE-2 GRAPHICS

Screen Modes 1 and 2 are expanded-text modes. Mode 1 expands the characters to twice their Mode-0 width, and Mode 2 expands the characters to twice their Mode-0 width and height. Modes 1 and 2 both have a 4-line text window along the bottom of the screen that is devoted to text operations that are similar to those of the Mode-0 screen.

The normal procedure for setting up the expanded-text modes is by executing a GRAPHICS 1 or GRAPHICS 2 command. In both instances, the system will clear the graphics and text-window portions of the screen. A later discussion in this section describes how to set up versions that do not have a text window, and enter one of the expanded-text modes without clearing the screen. Initiating either expanded-text mode automatically opens IOCB 6 for the graphics portion of the screens.

Organization of the Mode-1 and Mode-2 Color Registers

Fig. 4-5 shows the organization of the ATARI's five color registers as applied to both Modes 1 and 2. The figure also indicates the register organization for the text-window portion of the screens.

TEXT WINDOW USAGE		EXPANDED-CHARACTER SCREEN
NOT USED	REGISTER 0 ADDRESS 708	UPPER-CASE LETTERS, NUMERALS AND PUNCTUATION
CHARACTER LUMINANCE	REGISTER 1 ADDRESS 709	LOWER-CASE LETTERS AND SPECIAL GRAPHICS
CHARACTER HUE AND BACKGROUND COLOR	REGISTER 2 ADDRESS 710	INVERSE-VIDEO VERSION OF REGISTER 0
NOT USED	REGISTER 3 ADDRESS 711	INVERSE-VIDEO VERSION OF REGISTER 1
BORDER COLOR	REGISTER 4 ADDRESS 712	BACKGROUND COLOR

Fig. 4-5. Organization of the color registers for Modes 1 and 2.

Screen Modes 1 and 2 can use all five registers, with register 4 determining the background color for the graphics portion of the screen (which, incidentally, is the same as the border color for the text window).

The graphics and text window also share registers 1 and 2, thus making it difficult to print a graphic from register 2 that has a color different from the text-window's background. Likewise, working with color register 1 in terms of graphics will influence the luminance of the characters in the text-window area.

From BASIC, the color registers can be set by applying the SETCOLOR statement or by POKEing the appropriate data directly into the registers.

The SETCOLOR statement works with the register color in terms of separate hue and luminance values:

SETCOLOR *reg,hue,lum*

where *reg* is the register number (0 through 4), and *hue* and *lum* values are specified according to Tables 4-2 and 4-3.

When using POKE statements to set the color registers, the *addr* portion of the statement refers to the address of the color register and the *data* portion refers to the overall color designation: $16 * \text{hue} + \text{lum}$. So in a general sense, a POKE statement for setting the color registers takes this form:

POKE *addr,16hue* + *lum****

Accessing the Character Set From Modes 1 and 2

The graphics portion of the screen for Modes 1 and 2 uses the same character set as Mode 1 and the text windows for Modes 1 and 2. When working in the expanded-character modes, however, it is not possible to access the entire character set at any given moment—only one-half or the other is available.

Unless directed otherwise, the system accesses only the first 64 characters shown in Table 4-5 (internal codes 0 through 63). That includes all of the commonly used numerals, punctuation and upper-case letters of the alphabet. Whenever it is necessary to access the second half of the character table (codes 64 through 127), the MSB of that address of that part of the ROM table must be POKEd into address 756.

NOTE: Unless you are using a custom character set (described later in this chapter), the following POKE functions are necessary only when printing text to the expanded-character portion of screen Modes 1 and 2:

Access the characters for internal codes 0 through 63 by executing:

POKE 756,224

Access the characters for internal codes 64 through 127 by executing:

POKE 756,226

It is not possible to print inverse versions of the expanded characters.

Working With the Mode-1 and -2 Column/Row Format

Figs. 4-6 and 4-7 show the column/row formats for the Mode-1 and Mode-2 screens, respectively. Both have a 40 column, 4-row text window, but the expanded-character portions are different. The Mode-1 screen uses a 20-column, 10-row format, while the Mode-2 screen uses a 20-column, 20-row format.

Although it is also possible to regard these screens from a RAM address format, the column/row format is more appropriate when working with BASIC statements that refer directly to col and row parameters and use the print-cursor features.

BASIC's PRINT statement provides the most convenient means for printing characters to both portions of the Mode-1 and -2 screens. The expanded-character and text-window portions use a different PRINT syntax and different cursor-position registers.

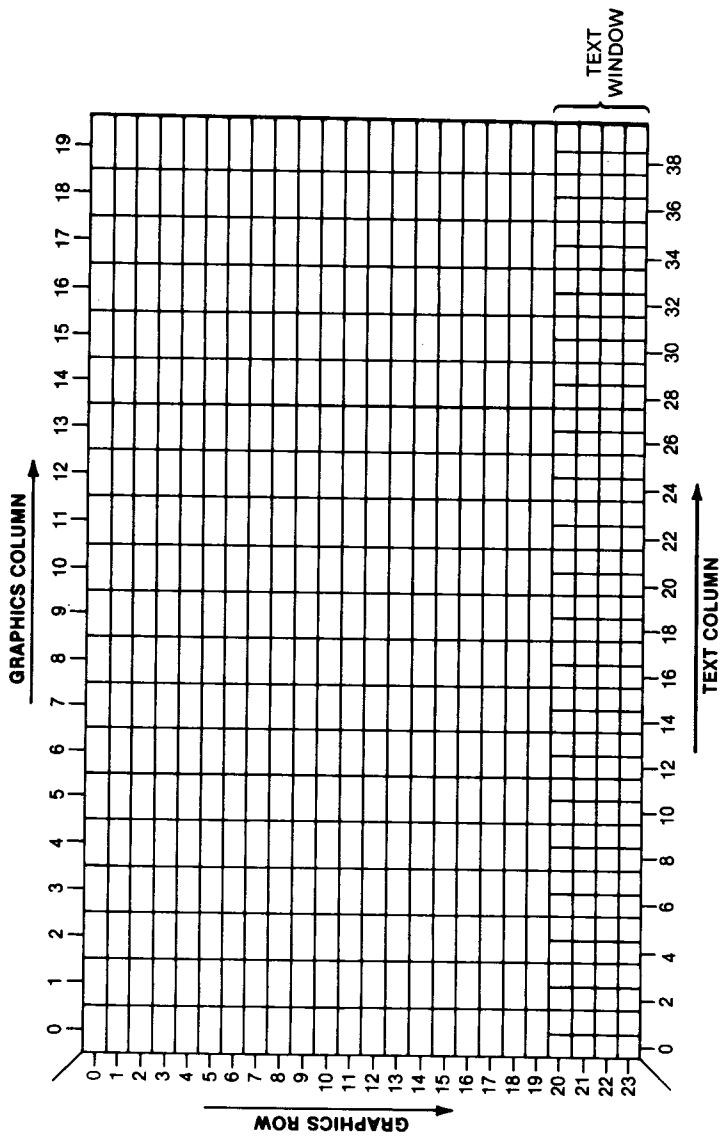


Fig. 4-6. Column/row format for screen Mode 1.

When working with split-screen Modes 1 and 2, ordinary PRINT statements refer to the text window and use the TXTCOL and TXTROW registers for keeping track of the text window cursor position (see Table 4-6).

But the printing operations for the expanded-character portion of the screen uses IOCB Channel 6. So any PRINT operation to that part of the screen must use a PRINT #6 statement. The general syntax is:

PRINT #6;*item*

where *item* is a valid PRINT constant or variable.

Printing operations to the expanded-character screen use cursor-position registers ROWCRS and COLCRS.

The expanded-character and text window portions of the screens thus use different cursor registers. Both sets of registers keep track of their respective cursor locations and carry out automatic operations, such as advancing the cursor after printing each character. The text window has a vertical scrolling feature, but the expanded-character portion of the screen does not. It is also possible to adjust the margin settings for the text window (as described for Mode-0 operations), but the expanded-character part of the screen has no margin feature at all.

Using a PRINT #6 statement to print characters directly to the expanded-character portion of the screen causes some potentially troublesome, but generally manageable, character and color effects. What is prescribed by a PRINT #6 statement doesn't necessarily appear that way on the expanded-character part of the screen. This is due to the fact that Modes 1 and 2 can access only half the internal character set at any given time (see Table 4-5). So if a PRINT #6 statement refers to some lower-case letters, and the system is working with the first half of the internal character set, those characters will be printed to the expanded-character screen as upper-case characters.

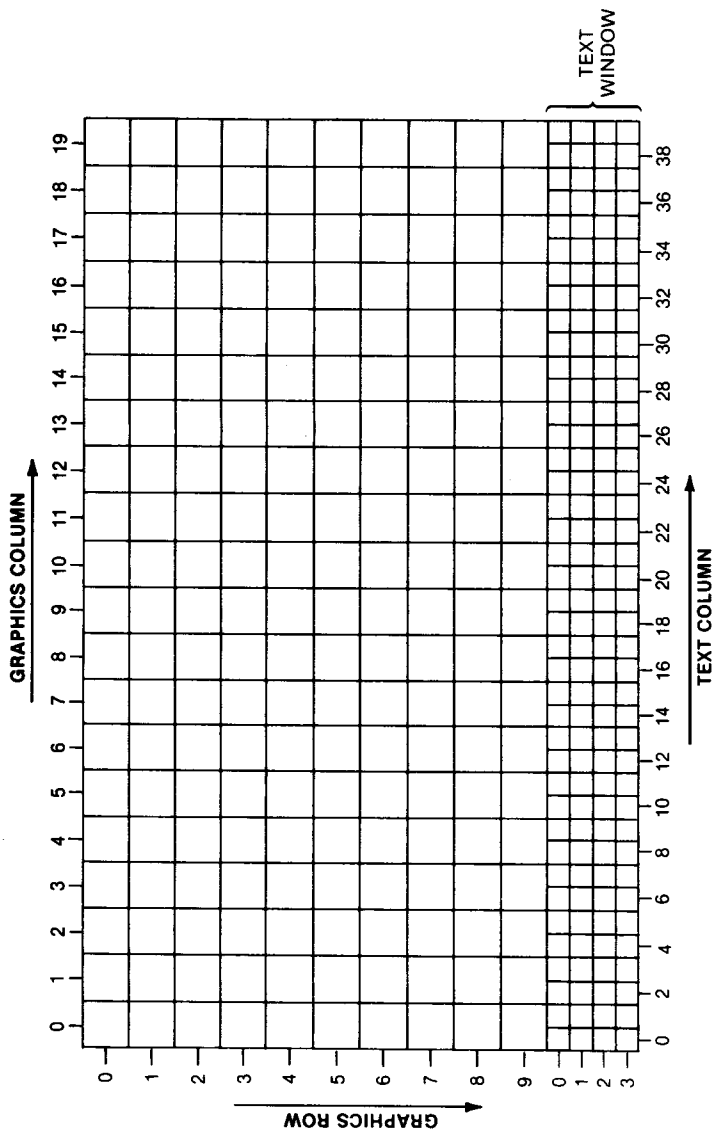


Fig. 4-7. Column/row format for screen Mode 2.

Furthermore, the color-register selection depends on the nature of the characters specified by a print #6 statement:

PRINT #6;*upper case* will print from color register 0

PRINT #6;*lower case* will print from color register 1

PRINT #6;*inverse upper case* will print from color register 2

PRINT #6;*inverse lower case* will print from color register 3

Using the POSITION Statement

The POSITION statement always affects cursor-position registers COLCRS and ROWCRS; and since these registers refer to the expanded-character cursor in Modes 1 and 2, it follows that the POSITION statement works only with that part of the screen. The text window is not affected by that statement.

When used with screen Modes 1 and 2, POSITION *col*, *row* statement refers directly to the screens' column/row format. In that context, the values assigned to *col* and *row* must be within the ranges specified for those screens in Figs. 4-6 and 4-7. For Mode 1 must be 0 through 19 for both parameters; and those assigned for Mode 2 must be 0 through 19 for the *col* parameter, and 0 through 9 for the *row* parameter.

POSITION is most useful for formatting text that is printed to the expanded-character portion of the screen.

As mentioned earlier, POSITION does not affect the formatting of text in the text window. It is possible to simulate a POSITION statement for that portion of the screen by POKEing *col* and *row* values directly into the text-window cursor registers, TXTCOL and TXTROW. The general syntax is:

```
POKE 656,row
POKE 657,col
```

where the text-window *row* values are between 0 and 3, and the *col* values are between 0 and 39.

Alternative Column/Row Techniques

Plotting expanded-character text to the Mode-1 and -2 screens is not limited to PRINT #6 statements; four statements that are normally regarded as purely graphics statements—COLOR, PLOT, DRAWTO, and XIO 18—can be useful, too. PLOT, DRAWTO, and XIO 18 deal with cursor-position registers OLDROW, OLDCOL, NEWROW and NEWCOL (see Table 4-6); and in screen Modes 1 and 2, those registers are dedicated to cursor operations in the expanded-character portions of the screen. Those statements are thus quite useful for expanded-text operations, but cannot be used for text-window operations.

































IOCB Channel 6 must be open in order to use them, but the execution of a GRAPHICS 1 or GRAPHICS 2 statement will do that job for you.

Used with the Mode-1 or Mode-2 screen, PLOT *col,row* prints a single character to the designated point on the expanded character portion of the screen. The character that is thus printed, and the color register it uses, is determined by the most recent COLOR statement. Table 4-7 summarizes the character and color register that is specified by a COLOR statement.

The table shows that each character can be printed from one of four color registers. Using the normal character set, for instance, a COLOR 97 will cause a subsequent PLOT statement to print an upper-case letter A from color register 2. But if the alternate internal character set is selected—by executing a POKE 756,226—the COLOR 97 statement will cause a subsequent PLOT operation to print a lower-case letter a from color register 2.

In screen Modes 1 and 2, then, the COLOR statement specifies both the character to be printed and the color register it is to use. Subsequent PLOT, DRAWTO, and XIO 18 statements will use that information.

































**Table 4-7. Character and Color-Register Codes
for the Color Statement in Screen Modes 1 and 2**

Color <i>value</i> and Register Used				Character Printed	
				norm	alt
32	0	160	128		
33	1	161	129		
34	2	162	130		
35	3	163	131		
36	4	164	132		
37	5	165	133		
38	6	166	134		
39	7	167	135		
40	8	168	136		
41	9	169	137		
42	10	170	138		
43	11	171	139		
44	12	172	140		
45	13	173	141		
46	14	174	142		
47	15	175	143		

































**Table 4-7—cont. Character and Color-Register Codes
for the Color Statement in Screen Modes 1 and 2**

Color value and Register Used				Character Printed	
				norm	alt
48	16	176	144	0	+
49	17	177	145	1	┐
50	18	178	146	2	—
51	19	179	147	3	+
52	20	180	148	4	●
53	21	181	149	5	▬
54	22	182	150	6	┌
55	23	183	151	7	└
56	24	184	152	8	┐
57	25	185	153	9	┌
58	26	186	154	÷	└
59	27	187	155	÷	┐
60	28	188	156	<	↑
61	29	189	157	=	↓
62	30	190	158	>	←
63	31	191	159	?	→

**Table 4-7—cont. Character and Color-Register Codes
for the Color Statement in Screen Modes 1 and 2**

Color <i>value</i> and Register Used				Character Printed	
				norm	alt
64	96	192	224		
65	97	193	225		
66	98	194	226		
67	99	195	227		
68	100	196	228		
69	101	197	229		
70	102	198	230		
71	103	199	231		
72	104	200	232		
73	105	201	233		
74	106	202	234		
75	107	203	235		
76	108	204	236		
77	109	205	237		
78	110	206	238		
79	111	207	239		

**Table 4-7—cont. Character and Color-Register Codes
for the Color Statement in Screen Modes 1 and 2**

Color <i>value</i> and Register Used				Character Printed	
				norm	alt
80	112	208	240		
81	113	209	241		
82	114	210	242		
83	115	211	243		
84	116	212	244		
85	117	213	245		
86	118	214	246		
87	119	215	247		
88	120	216	248		
89	121	217	249		
90	122	218	250		
91	123	219	251		
92	124	220	252		
93	125	221	253		
94	126	222	254		
95	127	223	255		

Normally, `DRAWTO col,row` will draw a line of color from a PLOT point to the designated column/row coordinate. The last-executed `COLOR` statement determines the color of that line. When working with the Mode-1 or Mode-2 screens, however, the parameter assigned to the `COLOR` statement determines the character and color to be plotted between a PLOT-specified place and the subsequent `DRAWTO`-specified place.

The `XIO 18` statement can fill a section of the screen with a specified character. For example:

```
10 GRAPHICS 2
20 PLOT 10,8:DRAWTO 10,0:DRAWTO 0,0:POSITION
   0,8
30 POKE 765,193
40 XIO 18,#6,0,0,"S:"
```

Program line 10 sets screen 2 (and opens IOCB #6 for screen operations), and line 20 defines the outline of a rectangular field on the screen. Line 30 `POKEs` the special `COLOR` code (Table 4-7) for the desired character, and line 40 executes the "fill" operation.

Using LOCATE, GET, and PUT in Modes 1 and 2

`LOCATE`, `GET`, and `PUT` also deal with the Mode-1 and -2 screens in a column/row format. Executing a `PUT #6,x` plots a character and uses a color register as specified in Table 4-7. The position is determined by the main print cursor registers, `ROWCRS` and `COLCRS`, and the execution of the statement advances the cursor position. In this context, the `PUT` statement is easier to use than the `COLOR/PLOT` combination.

The GET statement is the complement of PUT. Rather than plotting a specified character to the current cursor location, it returns a code value representing the character and color register (as represented in Table 4-7) for any character that might be printed there. The general form is:

GET #6,*numvar*

where *numvar* is a numerical variable that takes on the special code value.

The LOCATE function is quite similar to GET, but LOCATE does not operate according to the current cursor position. In fact, it operates entirely independent of it. Executing a statement of this general form:

LOCATE *col,row,numvar*

assigns the code number of any character printed to Mode-0 location *col,row* to numerical variable, *numvar*. The code format is the one that is shown in Table 4-7.

In screen Modes 1 and 2, the LOCATE statement works only with the expanded-character portion of the screen.

The Mode-1 and Mode-2 Screen RAM Formats

It is quite often helpful to view the Mode-1 and -2 screens in terms of the screen RAM addresses and data. Figs. 4-8 and 4-9 show the addressing ranges for each line on the screen.

This view of the Mode-1 and -2 screens facilitate the use of POKE and PEEK statements and machine-language routines for text operations. (The versions in the Appendices show both decimal and hexadecimal address locations.)

Fig. 4-8. Screen RAM address format for Mode 1.

POKE and PEEK statements both refer to an address and a byte of data that is associated with that address. The address in this case refers to the absolute screen address locations shown in the table; the data refers to the internal character codes cited in Table 4-5 as modified by a 2-bit color-register selection scheme:

$$\text{POKE } addr, 16 * reg + char$$

where *addr* is a RAM address for the expanded-character portion of the screen, *reg* is the desired color register, and *char* is the internal character code, 0 through 63.

The fact that the internal character codes are limited to 0 through 63 implies that the statement works with half of the character set and, further, it cannot deal with inverse characters. Executing a POKE 756,226 sets up the alternate half of the character set, and executing a POKE 756,224 returns the system to the normal character set.

In this context, the following sort of PEEK statement returns the combined internal-code value and color register of the character residing at the specified address:

$$\text{PEEK}(addr)$$

It is possible to do direct PEEK and POKE operations to the text-window portions of the screen. Tables 4-8 and 4-9 indicate the RAM addressing range for that part of the screen. The data refers to the ATASCII character codes in the same way it does for Mode-0 screen operations.

Full-Screen Formats

Adding 16 to the graphics screen number brings up the screen without a text window. Executing a GRAPHICS 17, for example, brings up a full-screen version of Mode-1. Several seconds after the conclusion of a program that uses the full screen, the system automatically reverts to its split-screen version.

[illegible]

Fig. 4-9. Screen RAM address format for Mode 2.

**Table 4-8. Color Register Sequences
and Screen Data Bytes for 4-Pixel/Bit Screens
(Modes 3,5,7, and Their Variations)**

Color Register (Left — Right)				Data Byte	Color Register (Left — Right)				Data Byte
0	0	0	0	0	0	2	0	0	32
0	0	0	1	1	0	2	0	1	33
0	0	0	2	2	0	2	0	2	34
0	0	0	3	3	0	2	0	3	35
0	0	1	0	4	0	2	1	0	36
0	0	1	1	5	0	2	1	1	37
0	0	1	2	6	0	2	1	2	38
0	0	1	3	7	0	2	1	3	39
0	0	2	0	8	0	2	2	0	40
0	0	2	1	9	0	2	2	1	41
0	0	2	2	10	0	2	2	2	42
0	0	2	3	11	0	2	2	3	43
0	0	3	0	12	0	2	3	0	44
0	0	3	1	13	0	2	3	1	45
0	0	3	2	14	0	2	3	2	46
0	0	3	3	15	0	2	3	3	47
0	1	0	0	16	0	3	0	0	48
0	1	0	1	17	0	3	0	1	49
0	1	0	2	18	0	3	0	2	50
0	1	0	3	19	0	3	0	3	51
0	1	1	0	20	0	3	1	0	52
0	1	1	1	21	0	3	1	1	53
0	1	1	2	22	0	3	1	2	54
0	1	1	3	23	0	3	1	3	55
0	1	2	0	24	0	3	2	0	56
0	1	2	1	25	0	3	2	1	57
0	1	2	2	26	0	3	2	2	58
0	1	2	3	27	0	3	2	3	59
0	1	3	0	28	0	3	3	0	60
0	1	3	1	29	0	3	3	1	61
0	1	3	2	30	0	3	3	2	62
0	1	3	3	31	0	3	3	3	63

**Table 4-8—cont. Color Register Sequences
and Screen Data Bytes for 4-Pixel/Bit Screens
(Modes 3,5,7, and Their Variations)**

Color Register (Left — Right)				Data Byte	Color Register (Left — Right)				Data Byte
1	0	0	0	64	1	2	0	0	96
1	0	0	1	65	1	2	0	1	97
1	0	0	2	66	1	2	0	2	98
1	0	0	3	67	1	2	0	3	99
1	0	1	0	68	1	2	1	0	100
1	0	1	1	69	1	2	1	1	101
1	0	1	2	70	1	2	1	2	102
1	0	1	3	71	1	2	1	3	103
1	0	2	0	72	1	2	2	0	104
1	0	2	1	73	1	2	2	1	105
1	0	2	2	74	1	2	2	2	106
1	0	2	3	75	1	2	2	3	107
1	0	3	0	76	1	2	3	0	108
1	0	3	1	77	1	2	3	1	109
1	0	3	2	78	1	2	3	2	110
1	0	3	3	79	1	2	3	3	111
1	1	0	0	80	1	3	0	0	112
1	1	0	1	81	1	3	0	1	113
1	1	0	2	82	1	3	0	2	114
1	1	0	3	83	1	3	0	3	115
1	1	1	0	84	1	3	1	0	116
1	1	1	1	85	1	3	1	1	117
1	1	1	2	86	1	3	1	2	118
1	1	1	3	87	1	3	1	3	119
1	1	2	0	88	1	3	2	0	120
1	1	2	1	89	1	3	2	1	121
1	1	2	2	90	1	3	2	2	122
1	1	2	3	91	1	3	2	3	123
1	1	3	0	92	1	3	3	0	124
1	1	3	1	93	1	3	3	1	125
1	1	3	2	94	1	3	3	2	126
1	1	3	3	95	1	3	3	3	127

**Table 4-8—cont. Color Register Sequences
and Screen Data Bytes for 4-Pixel/Bit Screens
(Modes 3,5,7, and Their Variations)**

Color Register (Left — Right)				Data Byte	Color Register (Left — Right)				Data Byte
2	0	0	0	128	2	2	0	0	160
2	0	0	1	129	2	2	0	1	161
2	0	0	2	130	2	2	0	2	162
2	0	0	3	131	2	2	0	3	163
2	0	1	0	132	2	2	1	0	164
2	0	1	1	133	2	2	1	1	165
2	0	1	2	134	2	2	1	2	166
2	0	1	3	135	2	2	1	3	167
2	0	2	0	136	2	2	2	0	168
2	0	2	1	137	2	2	2	1	169
2	0	2	2	138	2	2	2	2	170
2	0	2	3	139	2	2	2	3	171
2	0	3	0	140	2	2	3	0	172
2	0	3	1	141	2	2	3	1	173
2	0	3	2	142	2	2	3	2	174
2	0	3	3	143	2	2	3	3	175
2	1	0	0	144	2	3	0	0	176
2	1	0	1	145	2	3	0	1	177
2	1	0	2	146	2	3	0	2	178
2	1	0	3	147	2	3	0	3	179
2	1	1	0	148	2	3	1	0	180
2	1	1	1	149	2	3	1	1	181
2	1	1	2	150	2	3	1	2	182
2	1	1	3	151	2	3	1	3	183
2	1	2	0	152	2	3	2	0	184
2	1	2	1	153	2	3	2	1	185
2	1	2	2	154	2	3	2	2	186
2	1	2	3	155	2	3	2	3	187
2	1	3	0	156	2	3	3	0	188
2	1	3	1	157	2	3	3	1	189
2	1	3	2	158	2	3	3	2	190
2	1	3	3	159	2	3	3	3	191

**Table 4-8—cont. Color Register Sequences
and Screen Data Bytes for 4-Pixel/Bit Screens
(Modes 3,5,7, and Their Variations)**

Color Register (Left — Right)				Data Byte	Color Register (Left — Right)				Data Byte
3	0	0	0	192	3	2	0	0	224
3	0	0	1	193	3	2	0	1	225
3	0	0	2	194	3	2	0	2	226
3	0	0	3	195	3	2	0	3	227
3	0	1	0	196	3	2	1	0	228
3	0	1	1	197	3	2	1	1	229
3	0	1	2	198	3	2	1	2	230
3	0	1	3	199	3	2	1	3	231
3	0	2	0	200	3	2	2	0	232
3	0	2	1	201	3	2	2	1	233
3	0	2	2	202	3	2	2	2	234
3	0	2	3	203	3	2	2	3	235
3	0	3	0	204	3	2	3	0	236
3	0	3	1	205	3	2	3	1	237
3	0	3	2	206	3	2	3	2	238
3	0	3	3	207	3	2	3	3	239
3	1	0	0	208	3	3	0	0	240
3	1	0	1	209	3	3	0	1	241
3	1	0	2	210	3	3	0	2	242
3	1	0	3	211	3	3	0	3	243
3	1	1	0	212	3	3	1	0	244
3	1	1	1	213	3	3	1	1	245
3	1	1	2	214	3	3	1	2	246
3	1	1	3	215	3	3	1	3	247
3	1	2	0	216	3	3	2	0	248
3	1	2	1	217	3	3	2	1	249
3	1	2	2	218	3	3	2	2	250
3	1	2	3	219	3	3	2	3	251
3	1	3	0	220	3	3	3	0	252
3	1	3	1	221	3	3	3	1	253
3	1	3	2	222	3	3	3	2	254
3	1	3	3	223	3	3	3	3	255

**Table 4-9. Starting and Ending Addresses
for Each Row of the Mode-5 Screen RAM**

Row	Decimal		Row	Decimal	
	Start	End		Start	End
Row 0	39840	39859	Row 24	40320	40339
Row 1	39860	39879	Row 25	40340	40359
Row 2	39880	39899	Row 26	40360	40379
Row 3	39900	39919	Row 27	40380	40399
Row 4	39920	39939	Row 28	40400	40419
Row 5	39940	39959	Row 29	40420	40439
Row 6	39960	39979	Row 30	40440	40459
Row 7	39980	39999	Row 31	40460	40479
Row 8	40000	40019	Row 32	40480	40499
Row 9	40020	40039	Row 33	40500	40519
Row 10	40040	40059	Row 34	40520	40539
Row 11	40060	40079	Row 35	40540	40559
Row 12	40080	40099	Row 36	40560	40579
Row 13	40100	40119	Row 37	40580	40599
Row 14	40120	40139	Row 38	40600	40619
Row 15	40140	40159	Row 39	40620	40639
Row 16	40160	40179	Text window begins here		
Row 17	40180	40199			
Row 18	40200	40219	Row 0	40800	40839
Row 19	40220	40239	Row 1	40840	40879
Row 20	40240	40259	Row 2	40880	40919
Row 21	40260	40279	Row 3	40920	40959
Row 22	40280	40299			
Row 23	40300	40319			

Figs. 4-10 and 4-11 show the column/row formats for full-screen Modes 17 and 18, respectively. Those versions offer more rows of expanded-character space than is available with their split-screen counterparts.

Figs. 4-12 and 4-13 show full-screen Modes 17 and 18 in a screen RAM address format—one more suitable for PEEK, POKE, and machine-language operations.

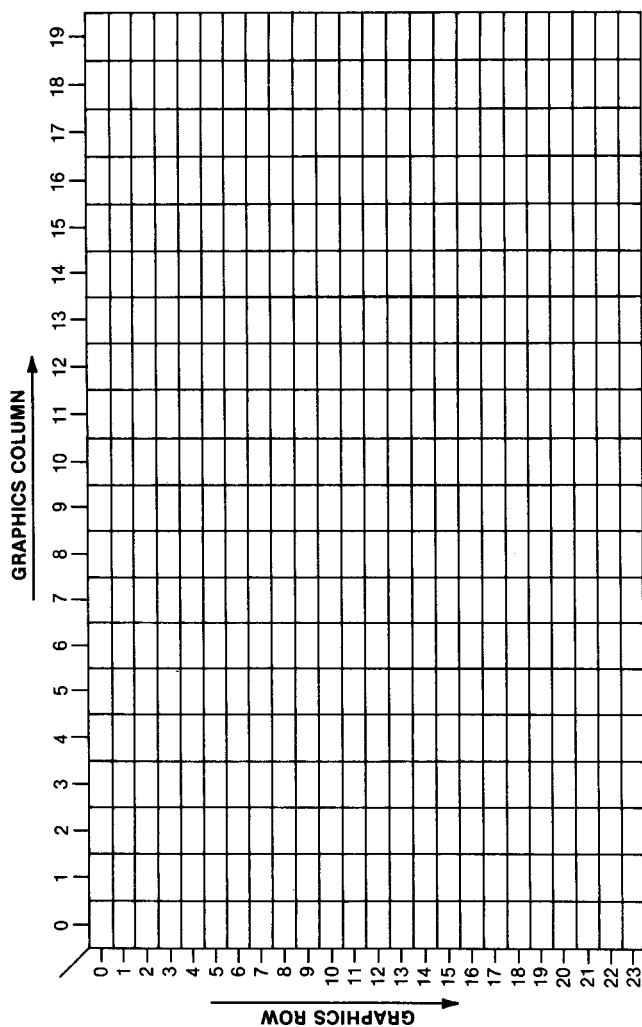


Fig. 4-10. Column/row format for Mode 17 (full-screen version of Mode 1).

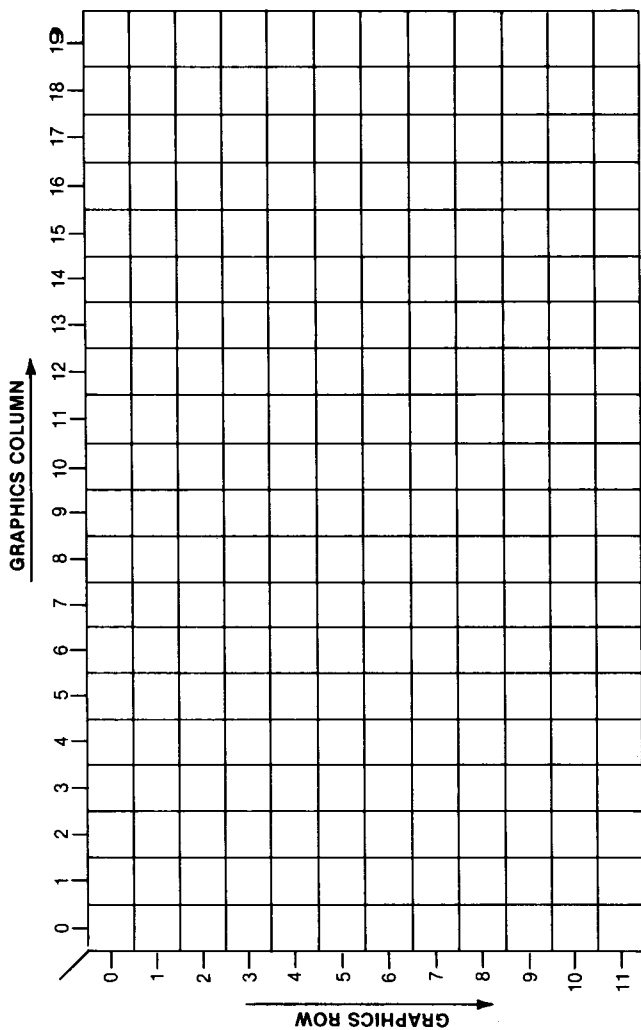


Fig. 4-11. Column/row format for Mode 18 (full-screen version of Mode 2).

[illegible]

Fig. 4-12. Screen RAM address format for Mode 17 (full-screen version of Mode 1).

CUSTOM CHARACTERS SETS FOR MODES 0, 1, AND 2

The foregoing discussions make numerous reference to the ATARI's ROM-based character set. Recall that the Mode-0 screen can access all 128 characters with no special programming, and that screen modes 1 and 2 can access it 64 characters at a time by POKEing appropriate values into register address 756.

It is possible, however, to construct custom character sets, load them into some unused RAM area, and then substitute them for ATARI's ROM-based set.

The character set must be bit mapped, with each character being 8 bytes long. Fig. 4-14 illustrates the procedure for generating the bit maps for four custom characters—the first four letters of the Hebrew alphabet, in this case.

Starting with a sheet of ordinary graph paper, construct the desired characters within an 8 x 8 character window. Then translate each square into a 0 or 1 binary bit; specify a 1 where a pixel of light is to be plotted, and a 0 where the background color is to appear. Finally, translate the resulting 8 bytes into their decimal counterparts. The series of decimal numbers are those that are to be inserted into the RAM area where your custom character set is to appear.

Actually, there will be five characters in this example, because the first character ought to be a blank—eight decimal zeros. The reason is that the computer will use the first character in the character set as the one for plotting the background; and you normally want the background to be blank.

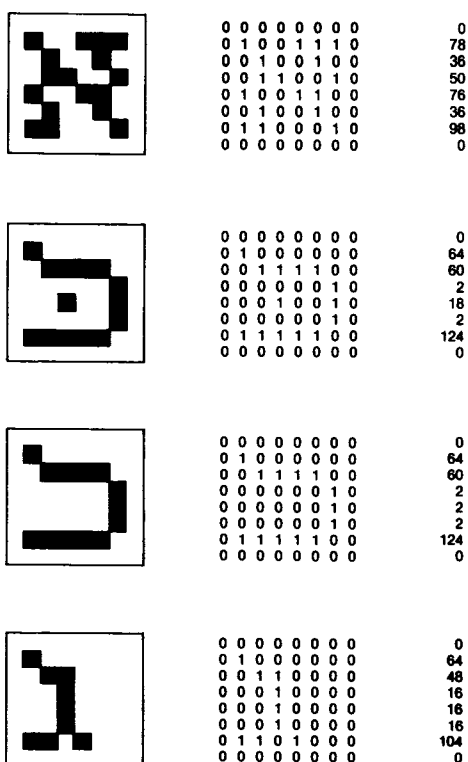


Fig. 4-14. Development of the bit map for several Hebrew characters to be used in a custom character set. The set must begin with an 8×8 bit map that is filled with zeros if the screen background is to be blank.

One good place to insert a custom, bit-mapped character set is at address 32768. (The standard ATARI character set begins at address 57344, or hexadecimal \$E000; but you cannot use that area because it is a ROM area.)

Here is a routine that will define the blank character and the four Hebrew characters, and insert them into RAM space at address 32768 (\$8000):

```
10 DATA 0,0,0,0
20 DATA 0,78,36,50,76,36,98,0
30 DATA 0,64,60,2,18,2,124,0
40 DATA 0,64,60,2,2,2,124,0
50 DATA 0,64,48,16,16,16,104,0
100 FOR N = 0 TO 39
110 READ A:POKE 32768+N,A
120 NEXT N
```

The DATA in lines 10 through 50 define the blank space and four Hebrew characters. Lines 100 through 120 then READ the data and POKE it into 40 successive RAM locations, beginning from address 32768. The custom character set is useless, however, unless you can print it to the screen.

A routine that lets you print the custom characters must begin by informing the system where the character set begins; and that is a matter of POKEing the MSB of the starting address into address 756—the same location that is used for specifying the first or second half of the standard character set.

The custom character set in this example begins at address 32768, or \$8000. The hexadecimal version of the MSB is \$80, and that translates into decimal 128. So in this instance, you specify the location of the custom character set with this sort of statement:

```
POKE 756,128
```

From that point on, the system will literally substitute your character set for the normal one.

Assuming, then, that you have executed a routine that loads the custom character set at address 32768, the following routine will print the blank character and four Hebrew characters to the Mode-0 screen:

```
200 GRAPHICS 0
210 SETCOLOR 1,0,14
220 POKE 756,128
230 FOR N = 0 TO 4
240 POKE 40500+N,N
250 NEXT N
```

All printing and expanded-character operations described for the ROM-based character set apply to custom character sets. The custom character set can have up to 128 different characters in it, and the system will access them in the same sequence that it accesses the ROM-based set. In Modes 1 and 2, you can access no more than 64 characters at a time; and whenever there is a need to access a different part of any character set, simply POKE the decimal version of the MSB of the starting address to location 756 (that, incidentally, allows you to mix custom and normal characters). And you can display inverse versions of a custom character set in screen Mode 0 by adding 128 to the usual internal character code number; likewise, you can use different color registers in Modes 1 and 2 by using the COLOR statement.

In short, a custom character set can be treated exactly like the normal ROM-based character set. The only difficult parts are designing and loading the custom character set into the system.

THE FOUR-COLOR MODES: 3, 5, AND 7

Graphics Modes 3, 5, 7, and their variations have a great deal in common, particularly with regard to the organization of the color registers and the screen data. The most obvious differences concern the size and number of pixels they use.

Working With the Color Registers

Fig. 4-15 shows the organization of the color registers for graphics Modes 3, 5, and 7. The text window uses the usual Mode-0 register format: the luminance of the characters is determined by register 1, the character hue and text-window background color is determined by register 2, and the text-window color is carried in register 4.

TEXT WINDOW		MODES 3,5,7
NOT USED	REGISTER 0 ADDRESS 708	COLOR 1
LUMINANCE OF CHARACTER	REGISTER 1 ADDRESS 709	COLOR 2
CHARACTER HUE AND BACKGROUND COLOR	REGISTER 2 ADDRESS 710	COLOR 3
NOT USED	REGISTER 3 ADDRESS 711	NOT USED
BORDER COLOR	REGISTER 4 ADDRESS 712	BACKGROUND COLOR 0

Fig. 4-15. Color register organization for Modes 3, 5, 7, and their variations.

The color-register diagram shows that the graphics and text window share three of the four color registers used in these graphics modes. For that reason, there is bound to be some interaction between the color environments for the graphics and text-window portions of the screen.

The colors assigned to any of these registers can be determined by executing the `SETCOLOR reg,hue,lum` statement, where *reg* refers to the standard color-register numbering format, and *hue* and *lum* are determined from Tables 4-2 and 4-3. It is also possible to specify the colors by executing `POKE addr,color` statements, where *addr* is the RAM address of the register and *color* is the overall color designation that is determined by the formula:

$$color = 16 * hue + lum$$

It is very important, however, to understand that the **COLOR** statement—the BASIC statement that refers some subsequent plotting operation to a color register—does not refer to the same register-numbering format as the **SETCOLOR** statement. The general **COLOR** *num* refers to color-register *num* in this fashion:

COLOR 0 — Register 4 (RAM address 712)
COLOR 1 — Register 0 (RAM address 708)
COLOR 2 — Register 1 (RAM address 709)
COLOR 3 — Register 2 (RAM address 710)

Column/Row Screen Formats and Graphics Operations

Figs. 4-16, 4-17, and 4-18 show the column/row format for screen Modes 3, 5, and 7, respectively. Notice that all of them include a 40-column, 4-row text window.

It is important to bear in mind the column/row sizes when using a good many of the BASIC graphics statements—attempting to work outside the column/row areas will bring up error messages.

The following BASIC graphics statements apply directly to these column/row formats:

POSITION *col, row*
PLOT *col, row*
DRAWTO *col, row*
LOCATE *col, row, x*

And these BASIC statements refer to the column/row formats in a less direct way—through the cursor-position registers (see Table 4-6):

PUT #6, *reg*
GET #6, *x*
PRINT #6;

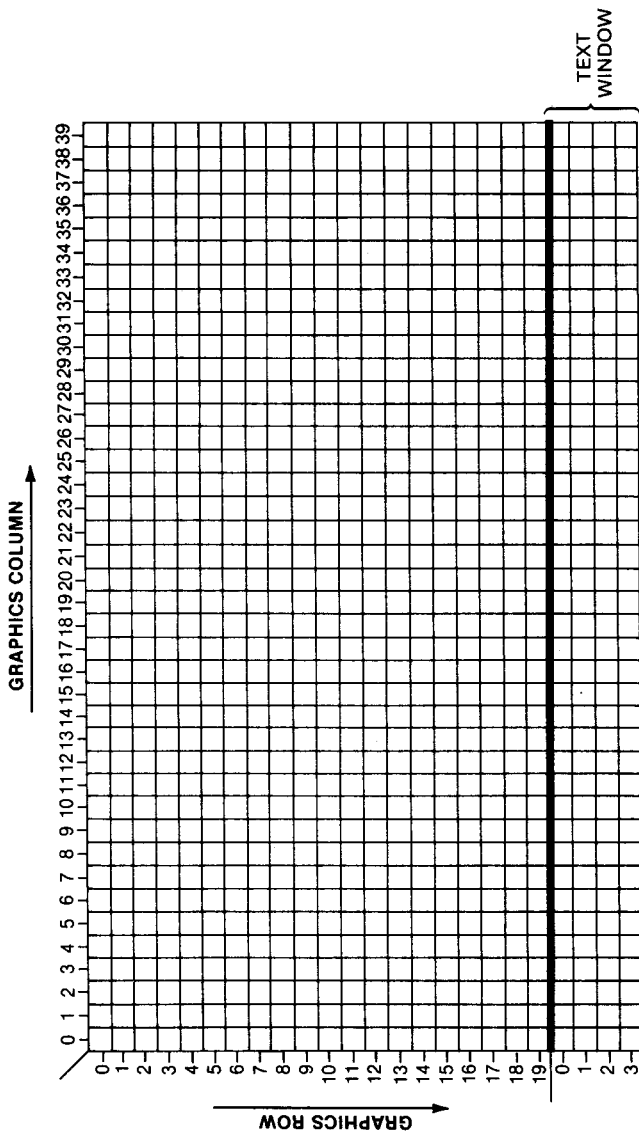


Fig. 4-16. Column/row format for the Mode-3 screen.

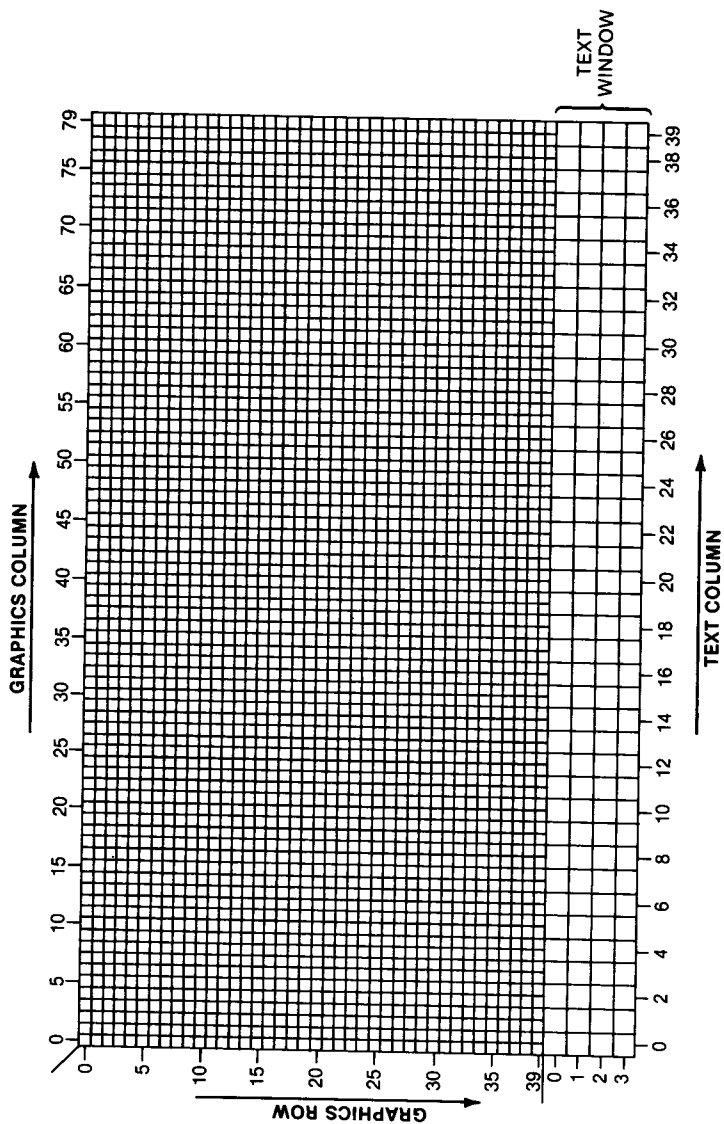


Fig. 4-17. Column/row format for the Mode-5 screen.

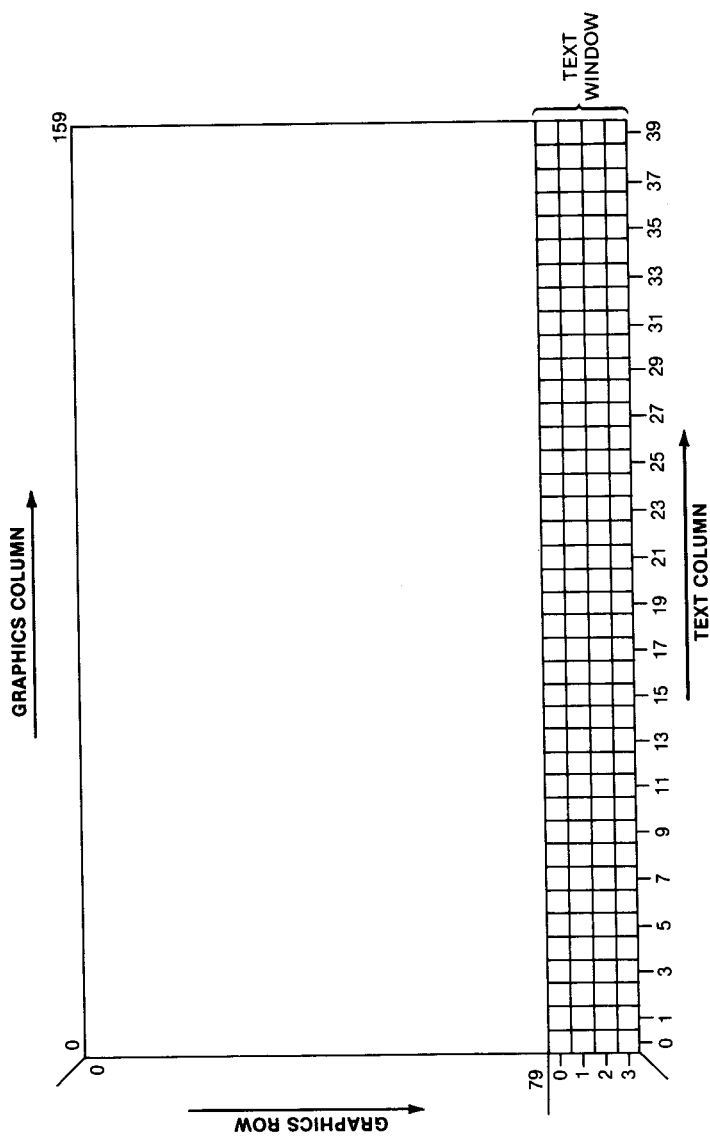


Fig. 4-18. Column/row format for the Mode-7 screen.

As in the case of the expanded-character modes, the PRINT statement refers to text-window coordinates as defined in register TXTROW and TXTCOL, and the XIO 18 function does a graphics fill operation when applied according to the following algorithm:

1. PLOT the lower right-hand pixel of the figure.
2. DRAWTO the upper right-hand pixel location.
3. DRAWTO the upper left-hand pixel location.
4. POSITION the lower right-hand pixel location.
5. POKE the color register to be used to address 765.
6. Execute XIO 18,#6,0,0,"S:".

Figs. 4-19, 4-20, and 4-21 show the full-screen versions of the 4-color modes that are called by graphics Modes 19, 21, and 23. Doing away with the text window adds more rows of graphics to the bottom of the screen.

The Screen RAM Address Formats and Operations

Fig. 4-22 shows the RAM addressing format for the Mode-3 screen. It is particularly important to notice that this 40-column screen uses 10 consecutive RAM address locations per row. The reason for this apparent discrepancy is that all 4-color modes use a single byte of data to specify the color registers for four consecutive column locations. It is thus possible to plot 800 different pixels (column/row format) with just 200 screen RAM locations (screen RAM format).

One byte of screen data covers four consecutive, horizontal pixel, or PLOT, locations on the screen; and the information carried within that byte specifies the color registers that are to be used for plotting its four graphics.

Consider a general POKE statement of this form:

POKE *addr,data*

In terms of the Mode-3 screen RAM format, *addr* references one of 200 groups of 4 pixels each. The *data* then sets a color for plotting each of those 4 pixels. Fig. 4-23 shows how the data byte is organized.

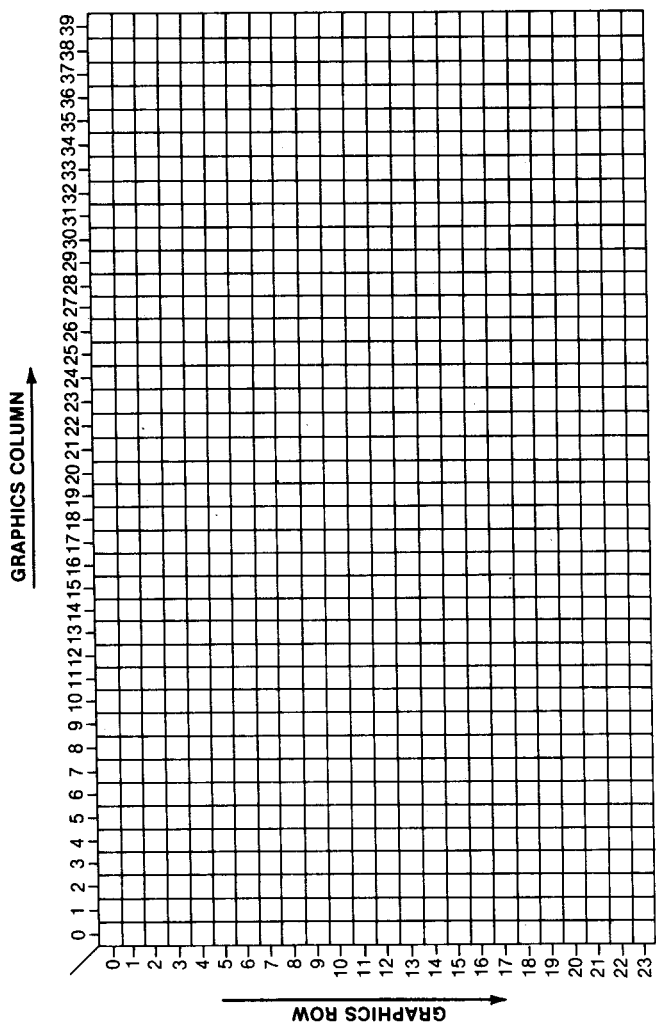


Fig. 4-19. Column/row format for the Mode-19 screen.

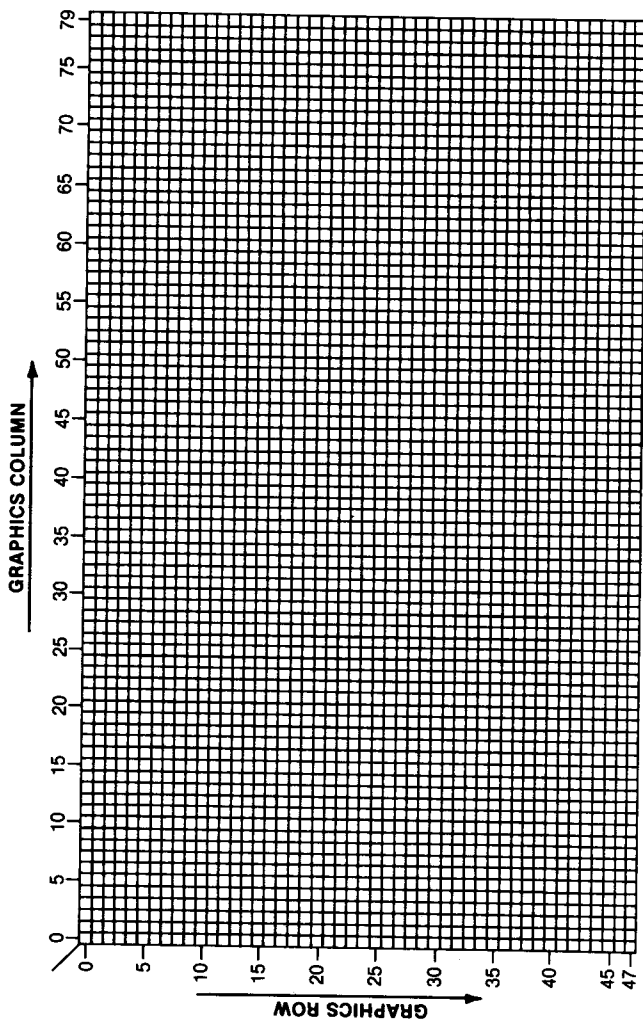


Fig. 4-20. Column/row format for the Mode-21 screen.

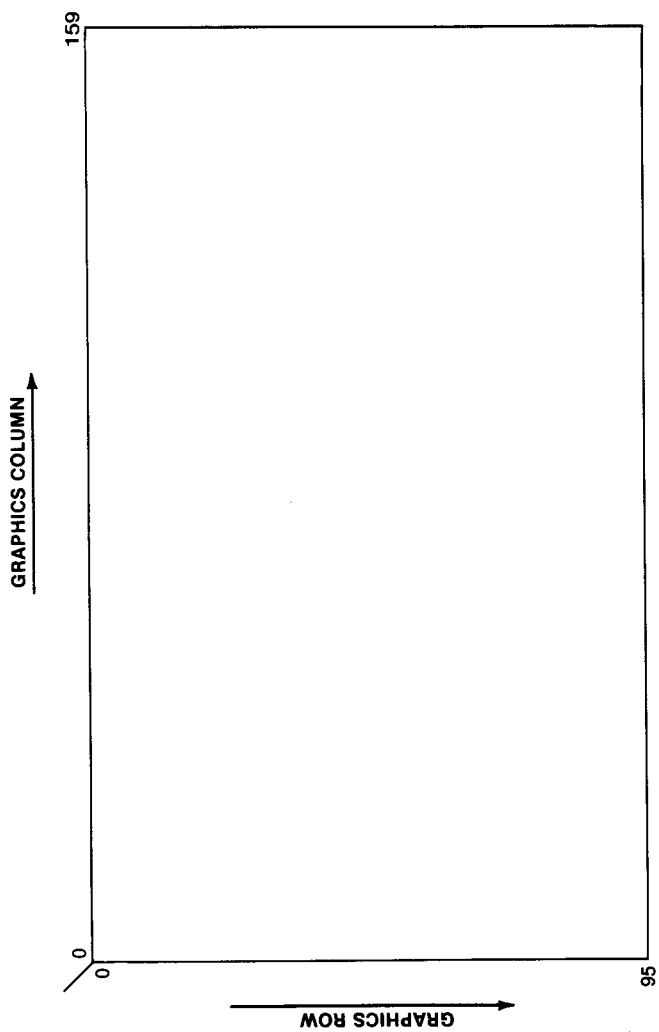


Fig. 4-21. Column/row format for the Mode-23 screen.

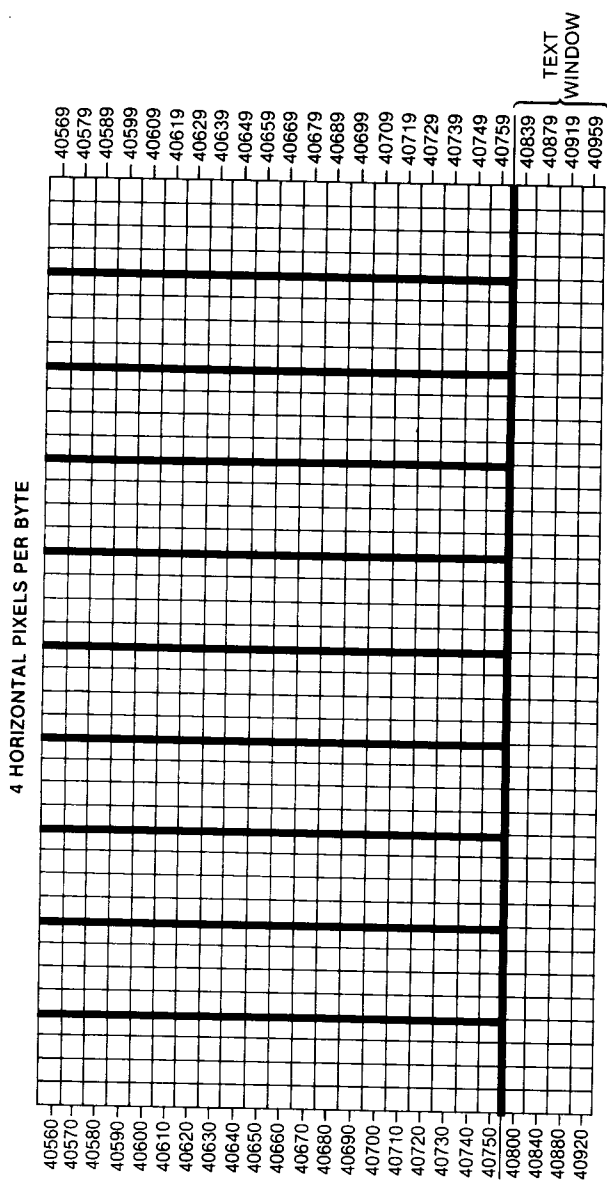


Fig. 4-22. Screen RAM address format for the Mode-3 screen.

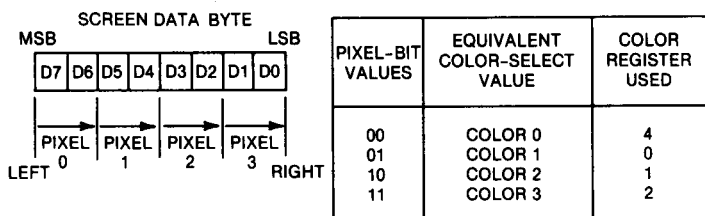


Fig. 4-23. Organization of screen data bytes for Modes 3, 5, 7, and their variations.

The data byte is divided into 4 pairs of bits. The two higher-order bits determine the color that is plotted by the leftmost pixel, bits D5 and D4 fix the color for the second pixel, bits D3 and D2 determine the color for the third pixel, and the two lower-order bytes determine the color for the rightmost pixel in the group.

In a manner of speaking, the *addr* portion of a POKE statement determines the location of the plotting operation within four pixels; and then the *data* portion of both fine-tunes the pixel positions and determines their color.

Now recall the *COLOR num* statement as it applies to the 4-color graphics modes. The *num* term can be a positive integer value between 0 and 3, inclusively; thereby making it possible to specify colors from one of four different color registers. The four *num* values for a *COLOR* statement relate directly to the values assigned to the *data* portion of a POKE statement.

The figure shows a 2-bit binary equivalent for each *COLOR* statement that can be used in 4-color graphics. If a pair of bytes assigned to one pixel happens to be 00, a POKE statement will plot that pixel position with a color that is equivalent to that of doing a *COLOR 0* statement.

Suppose, then, that you POKE this *data* byte into some Mode-3 screen address:

10 11 00 01

The leftmost pixel will be plotted with a color that is equivalent to doing a COLOR 2 statement (a color from register 1). The second pixel will take a color that is equivalent to COLOR 3; the third will be equivalent to doing COLOR 0; and the rightmost pixel will be plotted with a color that is equivalent to doing a COLOR 1 statement.

The following program plots four consecutive pixels at a single RAM address:

```
10 GRAPHICS 3
20 POKE 40664,109
```

The POKE address points to the middle of the graphics screen; and if you break down the data—decimal 109—into its binary counterpart, you will find it is:

01 10 11 01

And that suggests this COLOR sequence:

```
COLOR 1
COLOR 2
COLOR 3
COLOR 1
```

Each byte of POKE data refers to four consecutive, horizontal pixel locations and four color registers. I have just described how the data byte is organized, but it is clearly a troublesome procedure to implement. On the other hand, graphics programs using POKE statements can have the same level of resolution, but switch around the color-register designations, with far fewer programming statements than the more conventional COLOR/PLOT combinations.

Table 4-8 goes a long way toward simplifying the procedure for determining the value of a desired POKE byte. The table indicates four different COLOR-oriented *num* values that are fit into a data byte and plot to the screen in a left-to-right fashion. Thus, if you want to POKE four consecutive pixels that use the equivalent of COLOR 1, COLOR 3, COLOR 3, COLOR 2, simply find the sequence 1 3 3 2 in the table. The decimal data byte associated with that combination—126 in this case—is the corresponding data for the POKE statement.

The procedure applies equally well to all 4-color graphics screen formats.

Fig. 4-24 shows the screen RAM organization for the Mode-5 screen, and Table 4-9 shows the starting and ending addresses for each row, including those used for the text-window portion of the screen. Fig. 4-25 and Table 4-10 show the same information as it relates to the Mode-7 screen.

Figs. 4-26 through 4-28, and Tables 4-11 through 4-13 show the RAM screen organization and corresponding start-and-end addresses for each row of screen Modes 19, 21, and 23.

TWO-COLOR MODES 4 AND 6

Screen Modes 4 and 6 are 2-color, high-resolution graphics screens. They use a color-register scheme and data-byte format that is different from the 4-color screens.

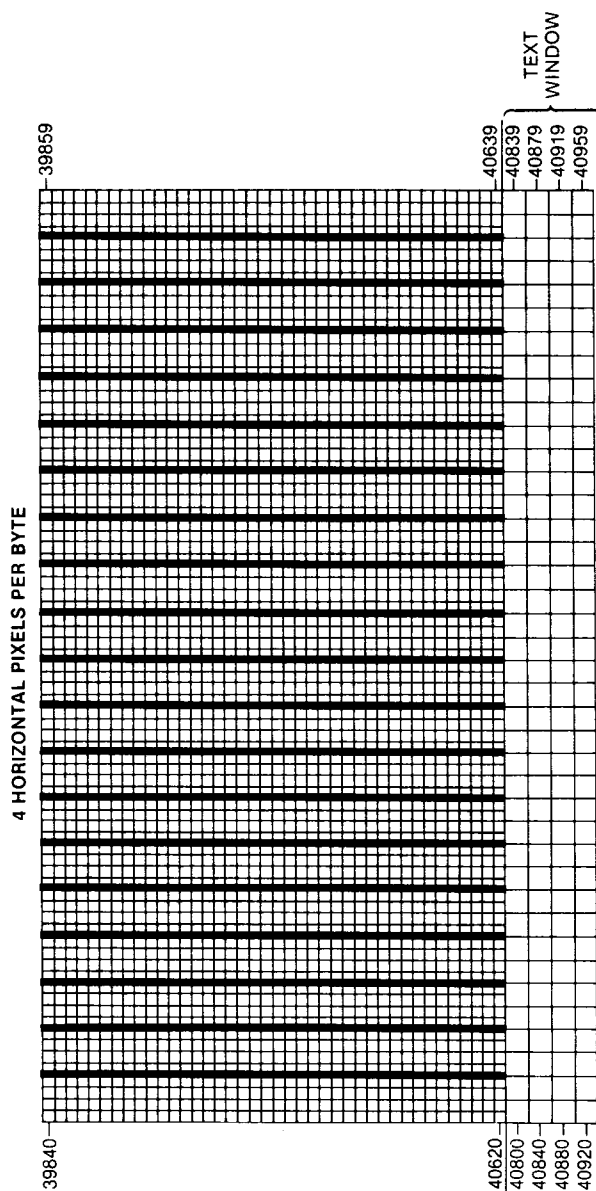


Fig. 4-24. Screen RAM address format for the Mode-5 screen. See detailed listing of the addresses in Table 4-9.

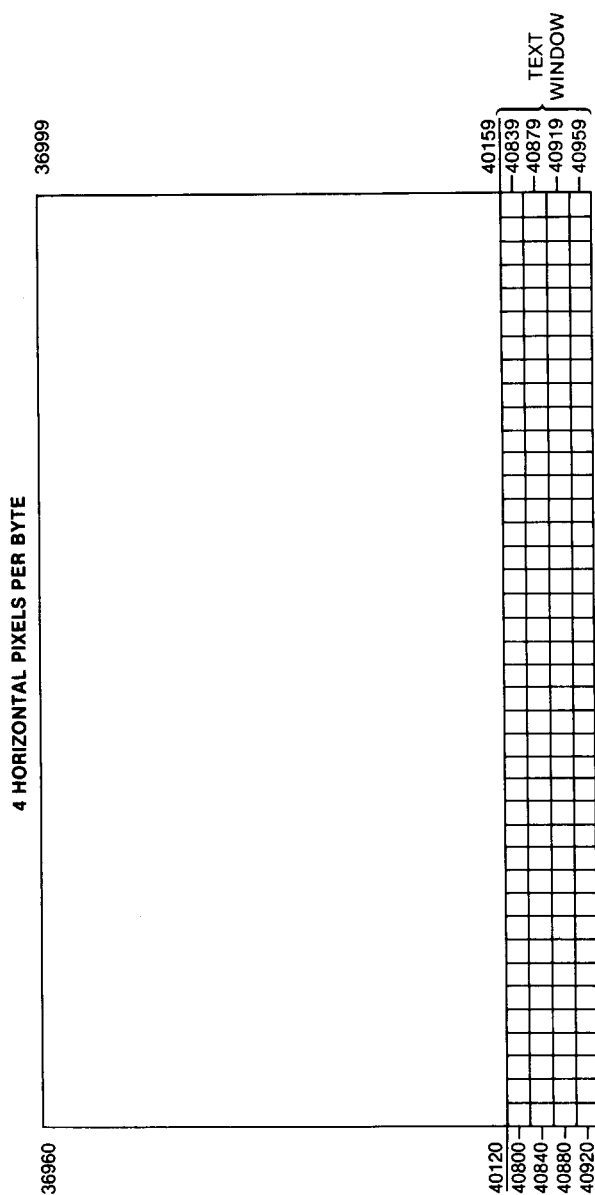


Fig. 4-25. Screen RAM address format for the Mode-7 screen. See a detailed listing of the addresses in Table 4-10.

**Table 4-10. Starting and Ending Addresses
for Each Row of the Mode-7 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 0	36960	36999	Row 44	38720	38759
Row 1	37000	37039	Row 45	38760	38799
Row 2	37040	37079	Row 46	38800	38839
Row 3	37080	37119	Row 47	38840	38879
Row 4	37120	37159	Row 48	38880	38919
Row 5	37160	37199	Row 49	38920	38959
Row 6	37200	37239	Row 50	38960	38999
Row 7	37240	37279	Row 51	39000	39039
Row 8	37280	37319	Row 52	39040	39079
Row 9	37320	37359	Row 53	39080	39119
Row 10	37360	37399	Row 54	39120	39159
Row 11	37400	37439	Row 55	39160	39199
Row 12	37440	37479	Row 56	39200	39239
Row 13	37480	37519	Row 57	39240	39279
Row 14	37520	37559	Row 58	39280	39319
Row 15	37560	37599	Row 59	39320	39359
Row 16	37600	37639	Row 60	39360	39399
Row 17	37640	37679	Row 61	39400	39439
Row 18	37680	37719	Row 62	39440	39479
Row 19	37720	37759	Row 63	39480	39519
Row 20	37760	37799	Row 64	39520	39559
Row 21	37800	37839	Row 65	39560	39599
Row 22	37840	37879	Row 66	39600	39639
Row 23	37880	37919	Row 67	39640	39679
Row 24	37920	37959	Row 68	39680	39719
Row 25	37960	37999	Row 69	39720	39759
Row 26	38000	38039	Row 70	39760	39799
Row 27	38040	38079	Row 71	39800	39839
Row 28	38080	38119	Row 72	39840	39879
Row 29	38120	38159	Row 73	39880	39919
Row 30	38160	38199	Row 74	39920	39959
Row 31	38200	38239	Row 75	39960	39999
Row 32	38240	38279	Row 76	40000	40039
Row 33	38280	38319	Row 77	40040	40079
Row 34	38320	38359	Row 78	40080	40119
Row 35	38360	38399	Row 79	40120	40159
Row 36	38400	38439	Text window begins here		
Row 37	38440	38479			
Row 38	38480	38519	Row 0	40800	40839
Row 39	38520	38559	Row 1	40840	40879
Row 40	38560	38599	Row 2	40880	40919
Row 41	38600	38639	Row 3	40920	40959
Row 42	38640	38679			
Row 43	38680	38719			

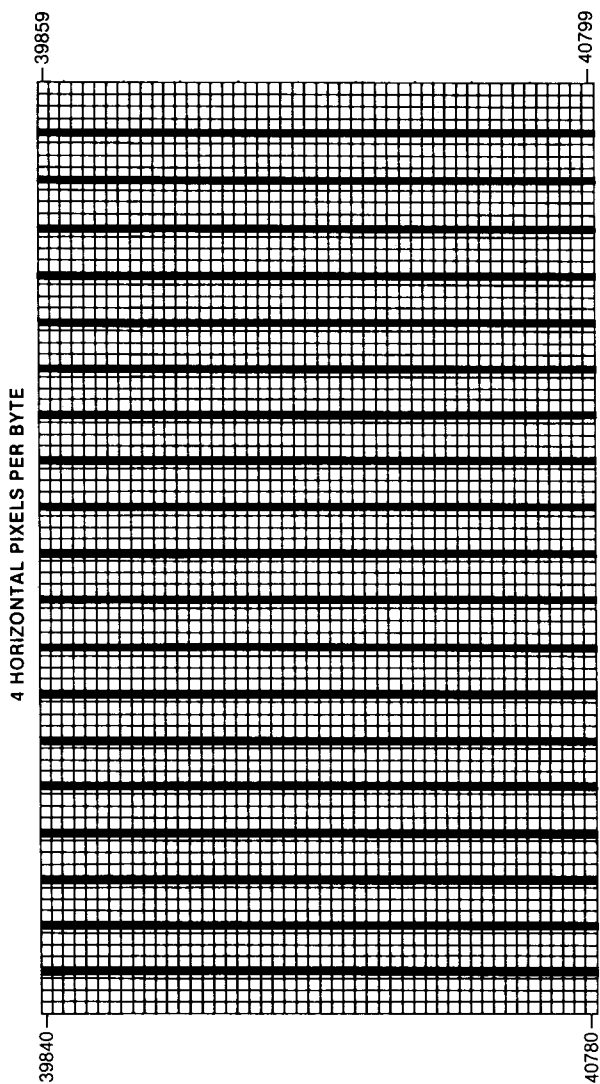


Fig. 4-27. Screen RAM address format for the Mode-21 screen. See a detailed listing of the addresses in Table 4-12.

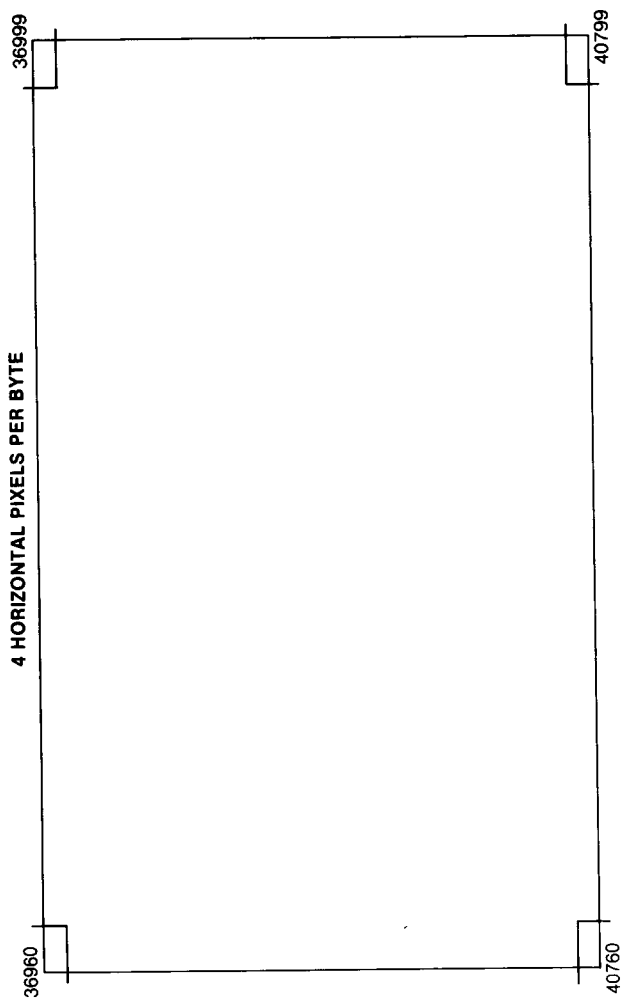


Fig. 4-28. Screen RAM address format for the Mode-23 screen. See a detailed listing of the addresses in Table 4-13.

Working With the Color Registers for Screen Modes 4 and 6

Fig. 4-29 shows the organization of the color registers for screen Modes 4, 6 and their variations. The text window uses the usually Mode-0 register format: the luminance of the characters is determined by register 1, the character hue and text-window background color is determined by register 2, and the text-window color is carried in register 4.

Unlike the 4-color screens, there is no troublesome sharing of color registers between the graphics portion of the screen and the text window. The graphics and text-window portions do share register 4 (graphics COLOR 0), but register 0 (graphics COLOR 1) is now for the exclusive use of graphics operations.

You can set the color for register 0 in one of two ways:

SETCOLOR 0,*hue*,*lum*
or
POKE 708,16**lum* + *hue*

where the *hue* and *lum* values can be determined from Tables 4-2 and 4-3, respectively. In either case, you can then access that register by doing a COLOR 1 statement.

The system forces you to use color register 4 for the graphics background. A SETCOLOR statement will refer to it as register 4, but a COLOR statement will refer to it in terms of COLOR 0. Although it is dedicated to the background color, color register 4 can be useful for deleting selected portions of previously drawn graphics.

Table 4-11. Starting and Ending Addresses for Each Row of the Mode-19 Screen RAM

Row	Address	
	Start	End
Row 0	40560	40569
Row 1	40570	40579
Row 2	40580	40589
Row 3	40590	40599
Row 4	40600	40609
Row 5	40610	40619
Row 6	40620	40629
Row 7	40630	40639
Row 8	40640	40649
Row 9	40650	40659
Row 10	40660	40669
Row 11	40670	40679
Row 12	40680	40689
Row 13	40690	40699
Row 14	40700	40709
Row 15	40710	40719
Row 16	40720	40729
Row 17	40730	40739
Row 18	40740	40749
Row 19	40750	40759
Row 20	40760	40769
Row 21	40770	40779
Row 22	40780	40789
Row 23	40790	40799

Column/Row Screen Formats and Operations for Modes 4 and 6

Figs. 4-30 and 4-31 show the column/row screen formats for graphics Modes 4 and 6. These formats are most appropriate when using PLOT, POSITION, DRAWTO, and LOCATE statements; and they are, incidentally, important when working with LOCATE, PUT #6, GET #6, XIO 18, and PRINT #6 statements. As in the case of the 4-color, split-screen modes, these feature a 40-column, 4-row text window that can be directly accessed with normal PRINT statements.

**Table 4-12. Starting and Ending Addresses
for Each Row of the Mode-21 Screen RAM**

Row	Decimal		Row	Decimal	
	Start	End		Start	End
Row 0	39840	39859	Row 24	40320	40339
Row 1	39860	39879	Row 25	40340	40359
Row 2	39880	39899	Row 26	40360	40379
Row 3	39900	39919	Row 27	40380	40399
Row 4	39920	39939	Row 28	40400	40419
Row 5	39940	39959	Row 29	40420	40439
Row 6	39960	39979	Row 30	40440	40459
Row 7	39980	39999	Row 31	40460	40479
Row 8	40000	40019	Row 32	40480	40499
Row 9	40020	40039	Row 33	40500	40519
Row 10	40040	40059	Row 34	40520	40539
Row 11	40060	40079	Row 35	40540	40559
Row 12	40080	40099	Row 36	40560	40579
Row 13	40100	40119	Row 37	40580	40599
Row 14	40120	40139	Row 38	40600	40619
Row 15	40140	40159	Row 39	40620	40639
Row 16	40160	40179	Row 40	40640	40659
Row 17	40180	40199	Row 41	40660	40679
Row 18	40200	40219	Row 42	40680	40699
Row 19	40220	40239	Row 43	40700	40719
Row 20	40240	40259	Row 44	40720	40739
Row 21	40260	40279	Row 45	40740	40759
Row 22	40280	40299	Row 46	40760	40779
Row 23	40300	40319	Row 47	40780	40799

**Table 4-13. Starting and Ending Addresses
for Each Row of the Mode-23 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 0	36960	36999	Row 36	38400	38439
Row 1	37000	37039	Row 37	38440	38479
Row 2	37040	37079	Row 38	38480	38519
Row 3	37080	37119	Row 39	38520	38559
Row 4	37120	37159	Row 40	38560	38599
Row 5	37160	37199	Row 41	38600	38639
Row 6	37200	37239	Row 42	38640	38679
Row 7	37240	37279	Row 43	38680	38719
Row 8	37280	37319	Row 44	38720	38759
Row 9	37320	37359	Row 45	38760	38799
Row 10	37360	37399	Row 46	38800	38839
Row 11	37400	37439	Row 47	38840	38879
Row 12	37440	37479	Row 48	38880	38919
Row 13	37480	37519	Row 49	38920	38959
Row 14	37520	37559	Row 50	38960	38999
Row 15	37560	37599	Row 51	39000	39039
Row 16	37600	37639	Row 52	39040	39079
Row 17	37640	37679	Row 53	39080	39119
Row 18	37680	37719	Row 54	39120	39159
Row 19	37720	37759	Row 55	39160	39199
Row 20	37760	37799	Row 56	39200	39239
Row 21	37800	37839	Row 57	39240	39279
Row 22	37840	37879	Row 58	39280	39319
Row 23	37880	37919	Row 59	39320	39359
Row 24	37920	37959	Row 60	39360	39399
Row 25	37960	37999	Row 61	39400	39439
Row 26	38000	38039	Row 62	39440	39479
Row 27	38040	38079	Row 63	39480	39519
Row 28	38080	38119	Row 64	39520	39559
Row 29	38120	38159	Row 65	39560	39599
Row 30	38160	38199	Row 66	39600	39639
Row 31	38200	38239	Row 67	39640	39679
Row 32	38240	38279	Row 68	39680	39719
Row 33	38280	38319	Row 69	39720	39759
Row 34	38320	38359	Row 70	39760	39799
Row 35	38360	38399	Row 71	39800	39839

**Table 4-13—cont. Starting and Ending Addresses
for Each Row of the Mode-23 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 72	39840	39879	Row 84	40320	40359
Row 73	39880	39919	Row 85	40360	40399
Row 74	39920	39959	Row 86	40400	40439
Row 75	39960	39999	Row 87	40440	40479
Row 76	40000	40039	Row 88	40480	40519
Row 77	40040	40079	Row 89	40520	40559
Row 78	40080	40119	Row 90	40560	40599
Row 79	40120	40159	Row 91	40600	40639
Row 80	40160	40199	Row 92	40640	40679
Row 81	40200	40239	Row 93	40680	40719
Row 82	40240	40279	Row 94	40720	40759
Row 83	40280	40319	Row 95	40760	40799

TEXT WINDOW		MODES 4 & 6
NOT USED	REGISTER 0 ADDRESS 708	COLOR 1
LUMINANCE OF CHARACTER	REGISTER 1 ADDRESS 709	NOT USED
CHARACTER HUE AND BACKGROUND COLOR	REGISTER 2 ADDRESS 710	NOT USED
NOT USED	REGISTER 3 ADDRESS 711	NOT USED
BORDER COLOR	REGISTER 4 ADDRESS 712	BACKGROUND AND COLOR 0

Fig. 4-29. Color register organization for Modes 4, 6, and their variations.

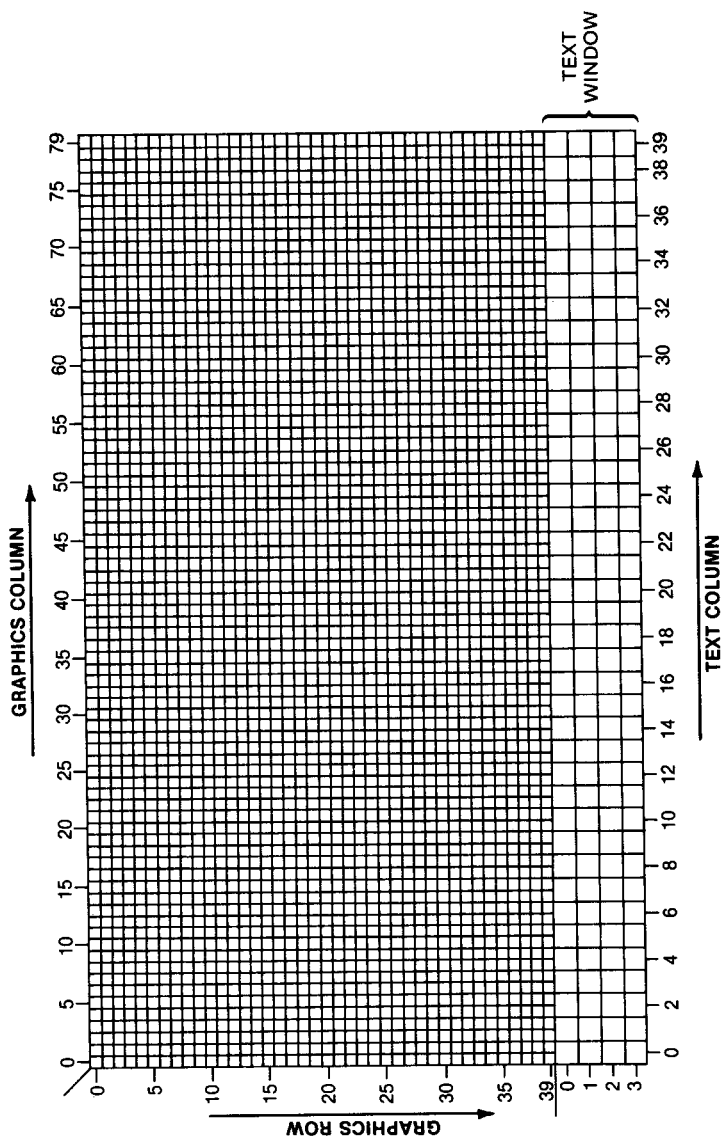


Fig. 4-30. Column/row format for the Mode-4 screen.

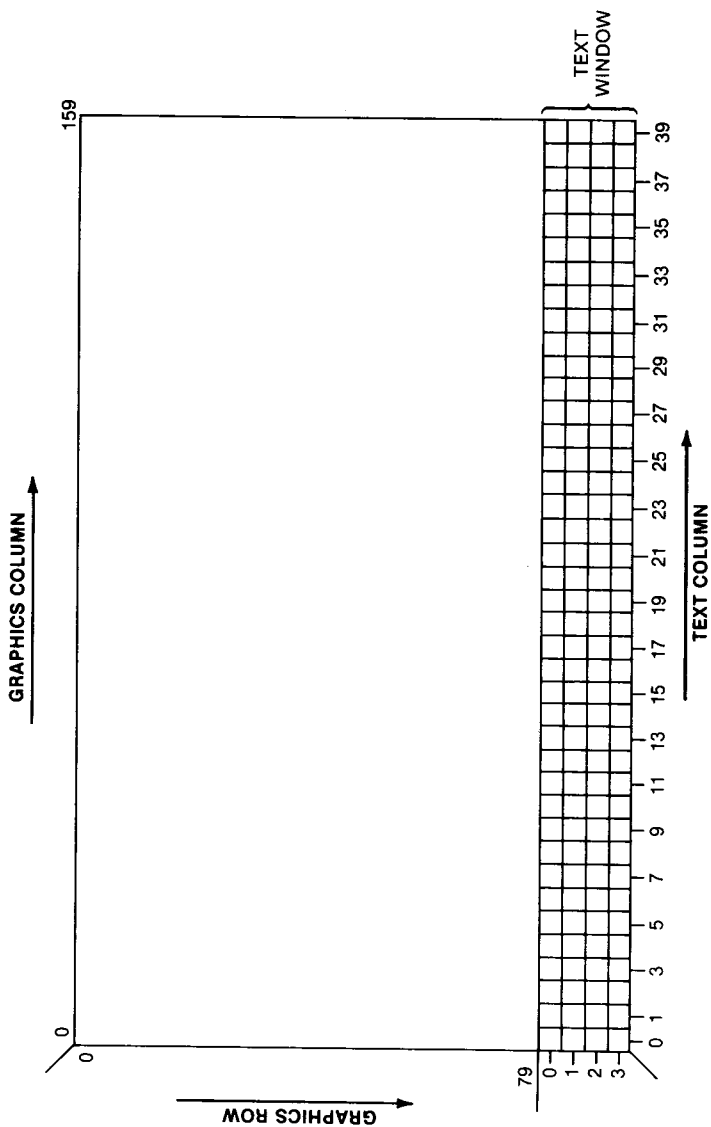


Fig. 4-31. Column/row format for the Mode-6 screen.

Figs. 4-32 and 4-33 show the column/row formats for screen Modes 20 and 22—the full-screen versions of Modes 4 and 6, respectively.

Screen RAM Formats and Operations for Modes 4 and 6

Figs. 4-34 through 4-37 illustrate some key RAM addresses for screen Modes 4, 6, 20, and 22. Tables 4-14 through 4-17 indicate the starting and ending addresses for each row. A comparison of these RAM-addressing formats with the corresponding column/row formats show an 8:1 ratio between the number of columns per row and the number of successive RAM addresses per row. Each byte of screen data, in other words, cites the color register to be used by 8 successive pixel locations.

Fig. 4-38 shows the organization of the data byte that is carried by each screen RAM address location. Each of the 8 bits carries a 0 or 1 that corresponds to a COLOR designation for Mode-4 and Mode-6 graphics.

If you POKE a data byte that is equivalent to using ones in all eight locations (decimal 255), the graphics system will respond by plotting eight consecutive pixels from the COLOR 1 register.

POKE a 170 into a screen RAM address and you will see four pixels separated by background color. The binary equivalent of 170 is:

1 0 1 0 1 0 1 0

And that will give alternate pixels a color that is different from the background.

Table 4-18 makes it easier to determine the data values whenever it is necessary to POKE it directly to the 2-color screens.

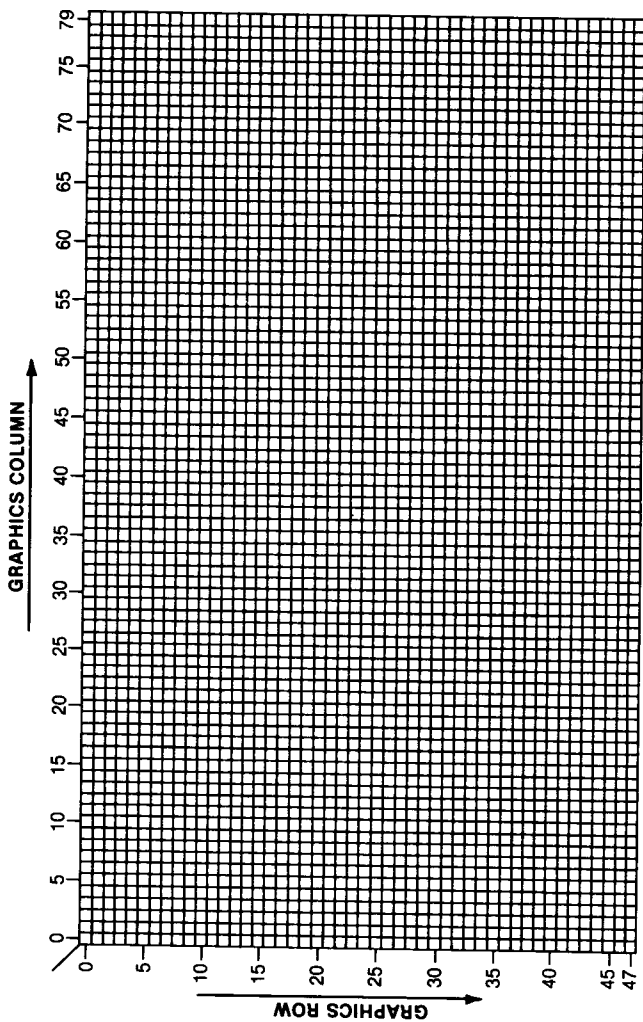


Fig. 4-32. Column/row format for the Mode-20 screen.

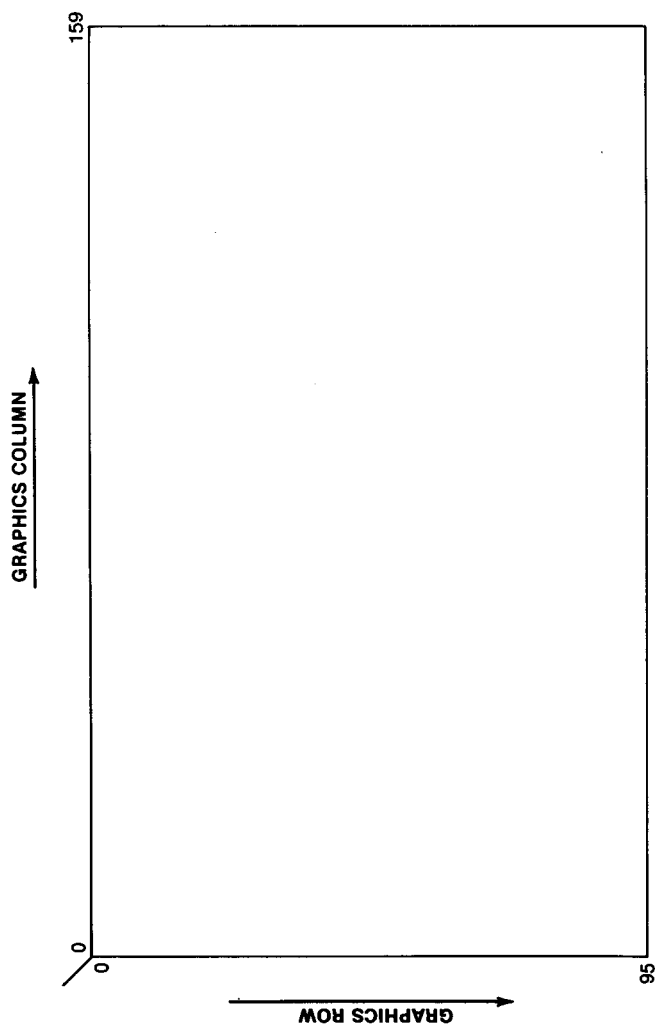


Fig. 4-33. Column/row format for the Mode-22 screen.

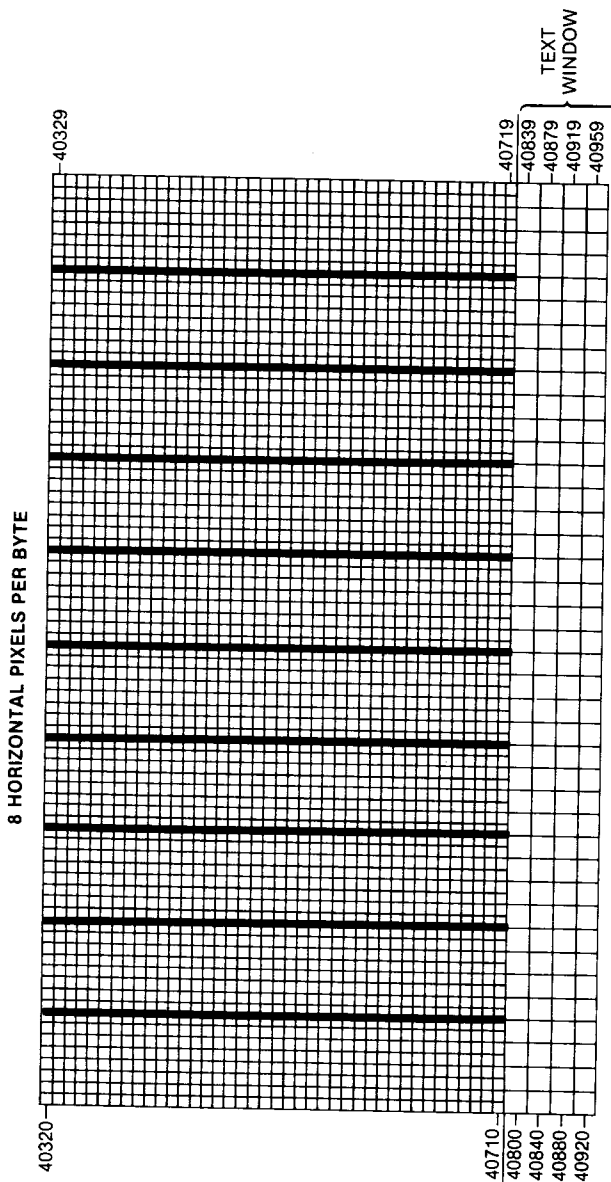


Fig. 4-34. Screen RAM address format for the Mode-4 screen. See a detailed listing of the addresses in Table 4-14.

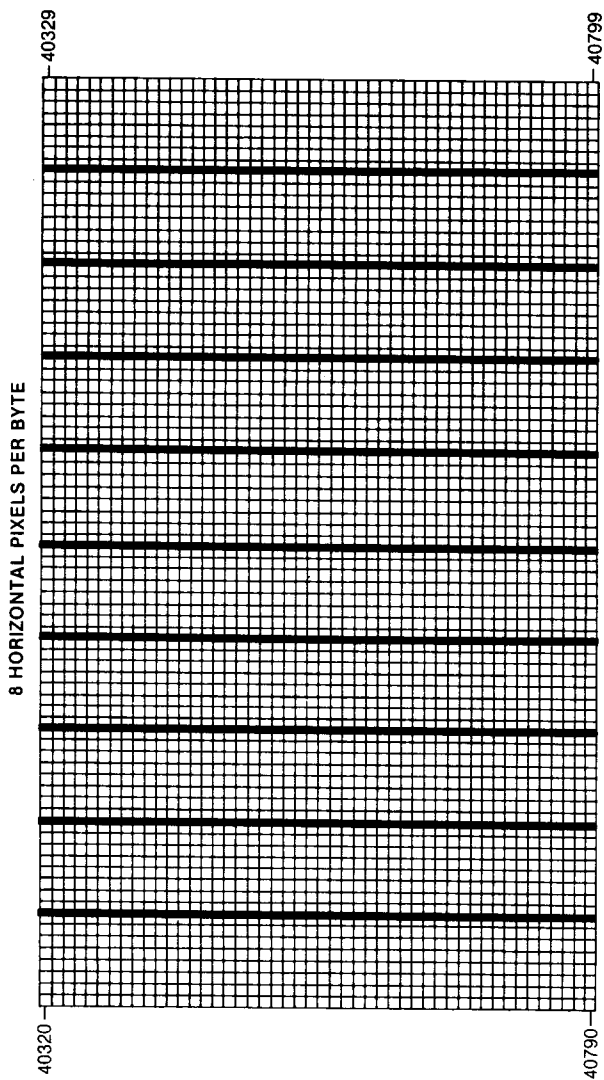


Fig. 4-36. Screen RAM address format for the Mode-20 screen. See a detailed listing of the addresses in Table 4-16.

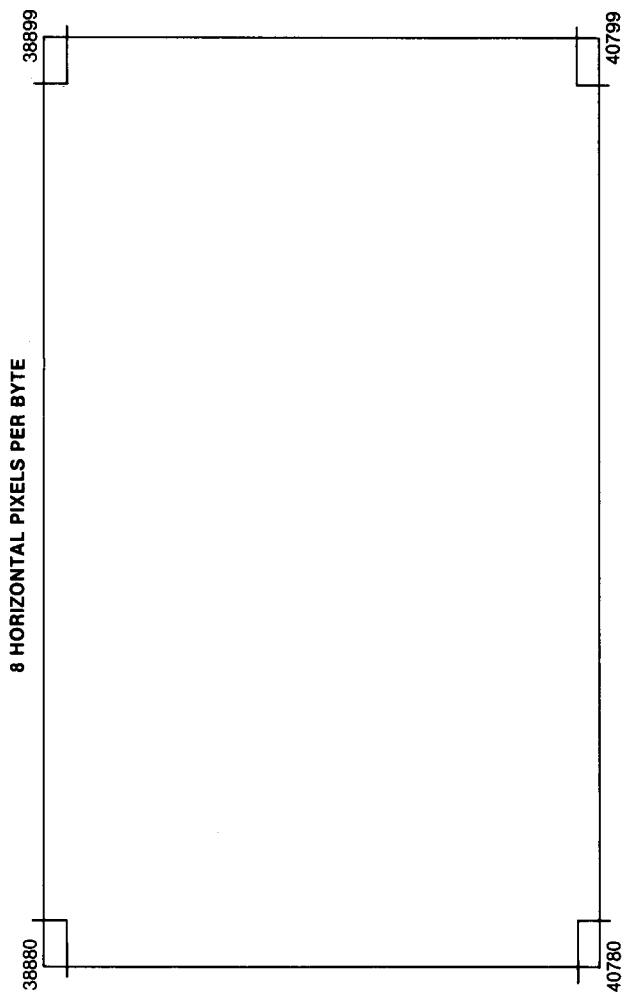


Fig. 4-37. Screen RAM address format for the Mode-22 screen. See a detailed listing of the addresses in Table 4-17.

**Table 4-14. Starting and Ending Addresses
for Each Row of the Mode-4 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 0	40320	40329	Row 24	40560	40569
Row 1	40330	40339	Row 25	40570	40579
Row 2	40340	40349	Row 26	40580	40589
Row 3	40350	40359	Row 27	40590	40599
Row 4	40360	40369	Row 28	40600	40609
Row 5	40370	40379	Row 29	40610	40619
Row 6	40380	40389	Row 30	40620	40629
Row 7	40390	40399	Row 31	40630	40639
Row 8	40400	40409	Row 32	40640	40649
Row 9	40410	40419	Row 33	40650	40659
Row 10	40420	40429	Row 34	40660	40669
Row 11	40430	40439	Row 35	40670	40679
Row 12	40440	40449	Row 36	40680	40689
Row 13	40450	40459	Row 37	40690	40699
Row 14	40460	40469	Row 38	40700	40709
Row 15	40470	40479	Row 39	40710	40719
Row 16	40480	40489	Text window begins here		
Row 17	40490	40499			
Row 18	40500	40509	Row 0	40800	40839
Row 19	40510	40519	Row 1	40840	40879
Row 20	40520	40529	Row 2	40880	40919
Row 21	40530	40539	Row 3	40920	40959
Row 22	40540	40549			
Row 23	40550	40559			

THE 2-COLOR MODE-8 SCREEN

The Mode-8 screen offers the highest resolution of all. It uses the same 8-bit screen-data format as Modes 4 and 6 (see Fig. 4-38), the same graphics commands, and the same general 2-color scheme. Aside from the higher level of resolution, the only difference between the Mode-8 and other 2-color modes is the organization of the two color registers.

**Table 4-15. Starting and Ending Addresses
for Each Row of the Mode-6 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 0	38880	38899	Row 44	39760	39779
Row 1	38900	38919	Row 45	39780	39799
Row 2	38920	38939	Row 46	39800	39819
Row 3	38940	38959	Row 47	39820	39839
Row 4	38960	38979	Row 48	39840	39859
Row 5	38980	38999	Row 49	39860	39879
Row 6	39000	39019	Row 50	39880	39899
Row 7	39020	39039	Row 51	39900	39919
Row 8	39040	39059	Row 52	39920	39939
Row 9	39060	39079	Row 53	39940	39959
Row 10	39080	39099	Row 54	39960	39979
Row 11	39100	39119	Row 55	39980	39999
Row 12	39120	39139	Row 56	40000	40019
Row 13	39140	39159	Row 57	40020	40039
Row 14	39160	39179	Row 58	40040	40059
Row 15	39180	39199	Row 59	40060	40079
Row 16	39200	39219	Row 60	40080	40099
Row 17	39220	39239	Row 61	40100	40119
Row 18	39240	39259	Row 62	40120	40139
Row 19	39260	39279	Row 63	40140	40159
Row 20	39280	39299	Row 64	40160	40179
Row 21	39300	39319	Row 65	40180	40199
Row 22	39320	39339	Row 66	40200	40219
Row 23	39340	39359	Row 67	40220	40239
Row 24	39360	39379	Row 68	40240	40259
Row 25	39380	39399	Row 69	40260	40279
Row 26	39400	39419	Row 70	40280	40299
Row 27	39420	39439	Row 71	40300	40319
Row 28	39440	39459	Row 72	40320	40339
Row 29	39460	39479	Row 73	40340	40359
Row 30	39480	39499	Row 74	40360	40379
Row 31	39500	39519	Row 75	40380	40399
Row 32	39520	39539	Row 76	40400	40419
Row 33	39540	39559	Row 77	40420	40439
Row 34	39560	39579	Row 78	40440	40459
Row 35	39580	39599	Row 79	40460	40479
Row 36	39600	39619	Text window begins here		
Row 37	39620	39639			
Row 38	39640	39659	Row 0	40800	40839
Row 39	39660	39679	Row 1	40840	40879
Row 40	39680	39699	Row 2	40880	40919
Row 41	39700	39719	Row 3	40920	40959
Row 42	39720	39739			
Row 43	39740	39759			

**Table 4-16. Starting and Ending Addresses
for Each Row of the Mode-20 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 0	40320	40329	Row 24	40560	40569
Row 1	40330	40339	Row 25	40570	40579
Row 2	40340	40349	Row 26	40580	40589
Row 3	40350	40359	Row 27	40590	40599
Row 4	40360	40369	Row 28	40600	40609
Row 5	40370	40379	Row 29	40610	40619
Row 6	40380	40389	Row 30	40620	40629
Row 7	40390	40399	Row 31	40630	40639
Row 8	40400	40409	Row 32	40640	40649
Row 9	40410	40419	Row 33	40650	40659
Row 10	40420	40429	Row 34	40660	40669
Row 11	40430	40439	Row 35	40670	40679
Row 12	40440	40449	Row 36	40680	40689
Row 13	40450	40459	Row 37	40690	40699
Row 14	40460	40469	Row 38	40700	40709
Row 15	40470	40479	Row 39	40710	40719
Row 16	40480	40489	Row 40	40720	40729
Row 17	40490	40499	Row 41	40730	40739
Row 18	40500	40509	Row 42	40740	40749
Row 19	40510	40519	Row 43	40750	40759
Row 20	40520	40529	Row 44	40760	40769
Row 21	40530	40539	Row 45	40770	40779
Row 22	40540	40549	Row 46	40780	40789
Row 23	40550	40559	Row 47	40790	40799

Fig. 4-39 shows that the Mode-8 colors refer to register 1 and 2. This can cause some difficulty when working with the text-window versions, because adjustments in the graphics color will affect the luminance of the text-window characters and both portions of the screen have the same background color.

When adjusting the Mode-8 graphics color, you have two options:

SETCOLOR 1,*hue*,*lum*
and
POKE 709,16**lum* + *hue*

Conveniently, you can PLOT that color by first executing a COLOR 1 statement.

**Table 4-17. Starting and Ending Addresses
for Each Row of the Mode-22 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 0	38880	38899	Row 36	39600	39619
Row 1	38900	38919	Row 37	39620	39639
Row 2	38920	38939	Row 38	39640	39659
Row 3	38940	38959	Row 39	39660	39679
Row 4	38960	38979	Row 40	39680	39699
Row 5	38980	38999	Row 41	39700	39719
Row 6	39000	39019	Row 42	39720	39739
Row 7	39020	39039	Row 43	39740	39759
Row 8	39040	39059	Row 44	39760	39779
Row 9	39060	39079	Row 45	39780	39799
Row 10	39080	39099	Row 46	39800	39819
Row 11	39100	39119	Row 47	39820	39839
Row 12	39120	39139	Row 48	39840	39859
Row 13	39140	39159	Row 49	39860	39879
Row 14	39160	39179	Row 50	39880	39899
Row 15	39180	39199	Row 51	39900	39919
Row 16	39200	39219	Row 52	39920	39939
Row 17	39220	39239	Row 53	39940	39959
Row 18	39240	39259	Row 54	39960	39979
Row 19	39260	39279	Row 55	39980	39999
Row 20	39280	39299	Row 56	40000	40019
Row 21	39300	39319	Row 57	40020	40039
Row 22	39320	39339	Row 58	40040	40059
Row 23	39340	39359	Row 59	40060	40079
Row 24	39360	39379	Row 60	40080	40099
Row 25	39380	39399	Row 61	40100	40119
Row 26	39400	39419	Row 62	40120	40139
Row 27	39420	39439	Row 63	40140	40159
Row 28	39440	39459	Row 64	40160	40179
Row 29	39460	39479	Row 65	40180	40199
Row 30	39480	39499	Row 66	40200	40219
Row 31	39500	39519	Row 67	40220	40239
Row 32	39520	39539	Row 68	40240	40259
Row 33	39540	39559	Row 69	40260	40279
Row 34	39560	39579	Row 70	40280	40299
Row 35	39580	39599	Row 71	40300	40319

**Table 4-17—cont. Starting and Ending Addresses
for Each Row of the Mode-22 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 72	40320	40339	Row 84	40560	40579
Row 73	40340	40359	Row 85	40580	40599
Row 74	40360	40379	Row 86	40600	40619
Row 75	40380	40399	Row 87	40620	40639
Row 76	40400	40419	Row 88	40640	40659
Row 77	40420	40439	Row 89	40660	40679
Row 78	40440	40459	Row 90	40680	40699
Row 79	40460	40479	Row 91	40700	40719
Row 80	40480	40499	Row 92	40720	40739
Row 81	40500	40519	Row 93	40740	40759
Row 82	40520	40539	Row 94	40760	40779
Row 83	40540	40559	Row 95	40780	40799

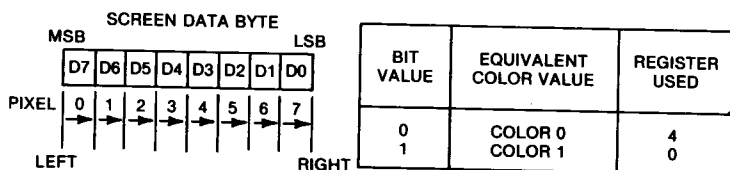


Fig. 4-38. Organization of screen data bytes for Modes 4, 6, and their variations.

But when it comes to the background color, you must SETCOLOR or POKE the information by designating register 2; and then use a COLOR 0 statement to access it.

**Table 4-18. Color Register Sequences and
Screen Data Bytes for 8-Pixel/Bit Screens
(Modes 4,6,8, and Their Variations)**

(Left ----- Right)								Data
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	1	1	3
0	0	0	0	0	1	0	0	4
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	1	0	6
0	0	0	0	0	1	1	1	7
0	0	0	0	1	0	0	0	8
0	0	0	0	1	0	0	1	9
0	0	0	0	1	0	1	0	10
0	0	0	0	1	0	1	1	11
0	0	0	0	1	1	0	0	12
0	0	0	0	1	1	0	1	13
0	0	0	0	1	1	1	0	14
0	0	0	0	1	1	1	1	15
0	0	0	1	0	0	0	0	16
0	0	0	1	0	0	0	1	17
0	0	0	1	0	0	1	0	18
0	0	0	1	0	0	1	1	19
0	0	0	1	0	1	0	0	20
0	0	0	1	0	1	0	1	21
0	0	0	1	0	1	1	0	22
0	0	0	1	0	1	1	1	23
0	0	0	1	1	0	0	0	24
0	0	0	1	1	0	0	1	25
0	0	0	1	1	0	1	0	26
0	0	0	1	1	0	1	1	27
0	0	0	1	1	1	0	0	28
0	0	0	1	1	1	0	1	29
0	0	0	1	1	1	1	0	30
0	0	0	1	1	1	1	1	31

**Table 4-18—cont. Color Register Sequences and
Screen Data Bytes for 8-Pixel/Bit Screens
(Modes 4,6,8, and Their Variations)**

(Left ----- Right)								Data
0	0	1	0	0	0	0	0	32
0	0	1	0	0	0	0	1	33
0	0	1	0	0	0	1	0	34
0	0	1	0	0	0	1	1	35
0	0	1	0	0	1	0	0	36
0	0	1	0	0	1	0	1	37
0	0	1	0	0	1	1	0	38
0	0	1	0	0	1	1	1	39
0	0	1	0	1	0	0	0	40
0	0	1	0	1	0	0	1	41
0	0	1	0	1	0	1	0	42
0	0	1	0	1	0	1	1	43
0	0	1	0	1	1	0	0	44
0	0	1	0	1	1	0	1	45
0	0	1	0	1	1	1	0	46
0	0	1	0	1	1	1	1	47
0	0	1	1	0	0	0	0	48
0	0	1	1	0	0	0	1	49
0	0	1	1	0	0	1	0	50
0	0	1	1	0	0	1	1	51
0	0	1	1	0	1	0	0	52
0	0	1	1	0	1	0	1	53
0	0	1	1	0	1	1	0	54
0	0	1	1	0	1	1	1	55
0	0	1	1	1	0	0	0	56
0	0	1	1	1	0	0	1	57
0	0	1	1	1	0	1	0	58
0	0	1	1	1	0	1	1	59
0	0	1	1	1	1	0	0	60
0	0	1	1	1	1	0	1	61
0	0	1	1	1	1	1	0	62
0	0	1	1	1	1	1	1	63

**Table 4-18—cont. Color Register Sequences and
Screen Data Bytes for 8-Pixel/Bit Screens
(Modes 4,6,8, and Their Variations)**

(Left ----- Right)								Data
0	1	0	0	0	0	0	0	64
0	1	0	0	0	0	0	1	65
0	1	0	0	0	0	1	0	66
0	1	0	0	0	0	1	1	67
0	1	0	0	0	1	0	0	68
0	1	0	0	0	1	0	1	69
0	1	0	0	0	1	1	0	70
0	1	0	0	0	1	1	1	71
0	1	0	0	1	0	0	0	72
0	1	0	0	1	0	0	1	73
0	1	0	0	1	0	1	0	74
0	1	0	0	1	0	1	1	75
0	1	0	0	1	1	0	0	76
0	1	0	0	1	1	0	1	77
0	1	0	0	1	1	1	0	78
0	1	0	0	1	1	1	1	79
0	1	0	1	0	0	0	0	80
0	1	0	1	0	0	0	1	81
0	1	0	1	0	0	1	0	82
0	1	0	1	0	0	1	1	83
0	1	0	1	0	1	0	0	84
0	1	0	1	0	1	0	1	85
0	1	0	1	0	1	1	0	86
0	1	0	1	0	1	1	1	87
0	1	0	1	1	0	0	0	88
0	1	0	1	1	0	0	1	89
0	1	0	1	1	0	1	0	90
0	1	0	1	1	0	1	1	91
0	1	0	1	1	1	0	0	92
0	1	0	1	1	1	0	1	93
0	1	0	1	1	1	1	0	94
0	1	0	1	1	1	1	1	95

**Table 4-18—cont. Color Register Sequences and
Screen Data Bytes for 8-Pixel/Bit Screens
(Modes 4,6,8, and Their Variations)**

(Left ----- Right)								Data
0	1	1	0	0	0	0	0	96
0	1	1	0	0	0	0	1	97
0	1	1	0	0	0	1	0	98
0	1	1	0	0	0	1	1	99
0	1	1	0	0	1	0	0	100
0	1	1	0	0	1	0	1	101
0	1	1	0	0	1	1	0	102
0	1	1	0	0	1	1	1	103
0	1	1	0	1	0	0	0	104
0	1	1	0	1	0	0	1	105
0	1	1	0	1	0	1	0	106
0	1	1	0	1	0	1	1	107
0	1	1	0	1	1	0	0	108
0	1	1	0	1	1	0	1	109
0	1	1	0	1	1	1	0	110
0	1	1	0	1	1	1	1	111
0	1	1	1	0	0	0	0	112
0	1	1	1	0	0	0	1	113
0	1	1	1	0	0	1	0	114
0	1	1	1	0	0	1	1	115
0	1	1	1	0	1	0	0	116
0	1	1	1	0	1	0	1	117
0	1	1	1	0	1	1	0	118
0	1	1	1	0	1	1	1	119
0	1	1	1	1	0	0	0	120
0	1	1	1	1	0	0	1	121
0	1	1	1	1	0	1	0	122
0	1	1	1	1	0	1	1	123
0	1	1	1	1	1	0	0	124
0	1	1	1	1	1	0	1	125
0	1	1	1	1	1	1	0	126
0	1	1	1	1	1	1	1	127

**Table 4-18—cont. Color Register Sequences and
Screen Data Bytes for 8-Pixel/Bit Screens
(Modes 4,6,8, and Their Variations)**

(Left ----- Right)								Data
1	0	0	0	0	0	0	0	128
1	0	0	0	0	0	0	1	129
1	0	0	0	0	0	1	0	130
1	0	0	0	0	0	1	1	131
1	0	0	0	0	1	0	0	132
1	0	0	0	0	1	0	1	133
1	0	0	0	0	1	1	0	134
1	0	0	0	0	1	1	1	135
1	0	0	0	1	0	0	0	136
1	0	0	0	1	0	0	1	137
1	0	0	0	1	0	1	0	138
1	0	0	0	1	0	1	1	139
1	0	0	0	1	1	0	0	140
1	0	0	0	1	1	0	1	141
1	0	0	0	1	1	1	0	142
1	0	0	0	1	1	1	1	143
1	0	0	1	0	0	0	0	144
1	0	0	1	0	0	0	1	145
1	0	0	1	0	0	1	0	146
1	0	0	1	0	0	1	1	147
1	0	0	1	0	1	0	0	148
1	0	0	1	0	1	0	1	149
1	0	0	1	0	1	1	0	150
1	0	0	1	0	1	1	1	151
1	0	0	1	1	0	0	0	152
1	0	0	1	1	0	0	1	153
1	0	0	1	1	0	1	0	154
1	0	0	1	1	0	1	1	155
1	0	0	1	1	1	0	0	156
1	0	0	1	1	1	0	1	157
1	0	0	1	1	1	1	0	158
1	0	0	1	1	1	1	1	159

**Table 4-18—cont. Color Register Sequences and
Screen Data Bytes for 8-Pixel/Bit Screens
(Modes 4,6,8, and Their Variations)**

(Left ----- Right)								Data
1	0	1	0	0	0	0	0	160
1	0	1	0	0	0	0	1	161
1	0	1	0	0	0	1	0	162
1	0	1	0	0	0	1	1	163
1	0	1	0	0	1	0	0	164
1	0	1	0	0	1	0	1	165
1	0	1	0	0	1	1	0	166
1	0	1	0	0	1	1	1	167
1	0	1	0	1	0	0	0	168
1	0	1	0	1	0	0	1	169
1	0	1	0	1	0	1	0	170
1	0	1	0	1	0	1	1	171
1	0	1	0	1	1	0	0	172
1	0	1	0	1	1	0	1	173
1	0	1	0	1	1	1	0	174
1	0	1	0	1	1	1	1	175
1	0	1	1	0	0	0	0	176
1	0	1	1	0	0	0	1	177
1	0	1	1	0	0	1	0	178
1	0	1	1	0	0	1	1	179
1	0	1	1	0	1	0	0	180
1	0	1	1	0	1	0	1	181
1	0	1	1	0	1	1	0	182
1	0	1	1	0	1	1	1	183
1	0	1	1	1	0	0	0	184
1	0	1	1	1	0	0	1	185
1	0	1	1	1	0	1	0	186
1	0	1	1	1	0	1	1	187
1	0	1	1	1	1	0	0	188
1	0	1	1	1	1	0	1	189
1	0	1	1	1	1	1	0	190
1	0	1	1	1	1	1	1	191

**Table 4-18—cont. Color Register Sequences and
Screen Data Bytes for 8-Pixel/Bit Screens
(Modes 4,6,8, and Their Variations)**

(Left ----- Right)								Data
1	1	0	0	0	0	0	0	192
1	1	0	0	0	0	0	1	193
1	1	0	0	0	0	1	0	194
1	1	0	0	0	0	1	1	195
1	1	0	0	0	1	0	0	196
1	1	0	0	0	1	0	1	197
1	1	0	0	0	1	1	0	198
1	1	0	0	0	1	1	1	199
1	1	0	0	1	0	0	0	200
1	1	0	0	1	0	0	1	201
1	1	0	0	1	0	1	0	202
1	1	0	0	1	0	1	1	203
1	1	0	0	1	1	0	0	204
1	1	0	0	1	1	0	1	205
1	1	0	0	1	1	1	0	206
1	1	0	0	1	1	1	1	207
1	1	0	1	0	0	0	0	208
1	1	0	1	0	0	0	1	209
1	1	0	1	0	0	1	0	210
1	1	0	1	0	0	1	1	211
1	1	0	1	0	1	0	0	212
1	1	0	1	0	1	0	1	213
1	1	0	1	0	1	1	0	214
1	1	0	1	0	1	1	1	215
1	1	0	1	1	0	0	0	216
1	1	0	1	1	0	0	1	217
1	1	0	1	1	0	1	0	218
1	1	0	1	1	0	1	1	219
1	1	0	1	1	1	0	0	220
1	1	0	1	1	1	0	1	221
1	1	0	1	1	1	1	0	222
1	1	0	1	1	1	1	1	223

**Table 4-18—cont. Color Register Sequences and
Screen Data Bytes for 8-Pixel/Bit Screens
(Modes 4,6,8, and Their Variations)**

(Left ----- Right)								Data
1	1	1	0	0	0	0	0	224
1	1	1	0	0	0	0	1	225
1	1	1	0	0	0	1	0	226
1	1	1	0	0	0	1	1	227
1	1	1	0	0	1	0	0	228
1	1	1	0	0	1	0	1	229
1	1	1	0	0	1	1	0	230
1	1	1	0	0	1	1	1	231
1	1	1	0	1	0	0	0	232
1	1	1	0	1	0	0	1	233
1	1	1	0	1	0	1	0	234
1	1	1	0	1	0	1	1	235
1	1	1	0	1	1	0	0	236
1	1	1	0	1	1	0	1	237
1	1	1	0	1	1	1	0	238
1	1	1	0	1	1	1	1	239
1	1	1	1	0	0	0	0	240
1	1	1	1	0	0	0	1	241
1	1	1	1	0	0	1	0	242
1	1	1	1	0	0	1	1	243
1	1	1	1	0	1	0	0	244
1	1	1	1	0	1	0	1	245
1	1	1	1	0	1	1	0	246
1	1	1	1	0	1	1	1	247
1	1	1	1	1	0	0	0	248
1	1	1	1	1	0	0	1	249
1	1	1	1	1	0	1	0	250
1	1	1	1	1	0	1	1	251
1	1	1	1	1	1	0	0	252
1	1	1	1	1	1	0	1	253
1	1	1	1	1	1	1	0	254
1	1	1	1	1	1	1	1	255

TEXT WINDOW USABLE		GRAPHICS MODE 8
NOT USED	REGISTER 0 ADDRESS 708	NOT USED
CHARACTER LUMINANCE	REGISTER 1 ADDRESS 709	COLOR 1
CHARACTER HUE AND BACKGROUND COLOR	REGISTER 2 ADDRESS 710	BACKGROUND
NOT USED	REGISTER 3 ADDRESS 711	NOT USED
BORDER COLOR	REGISTER 4 ADDRESS 712	BORDER COLOR

Fig. 4-39. Organization of color registers for the Mode-8 screen.

Fig. 4-40 shows the row/column format for the Mode-8 screen. Notice that it uses 320 columns (labeled 0 through 310) and 160 graphics rows (0 through 159). That figures out to be 51,200 pixel locations. Fig. 4-41, however, shows that the 8-pixel-per-byte scheme allows the system to do the job with just 6400 bytes of screen RAM.

Table 4-19 shows the first and last addresses for each row of the screen's RAM area.

Screen Mode 24 is a Mode-8 screen without the text window. Fig. 4-42 shows its 320-column, 192-row format, Fig. 4-43 indicates the screen RAM addressing range, and Table 4-20 details the RAM addressing range for each row.

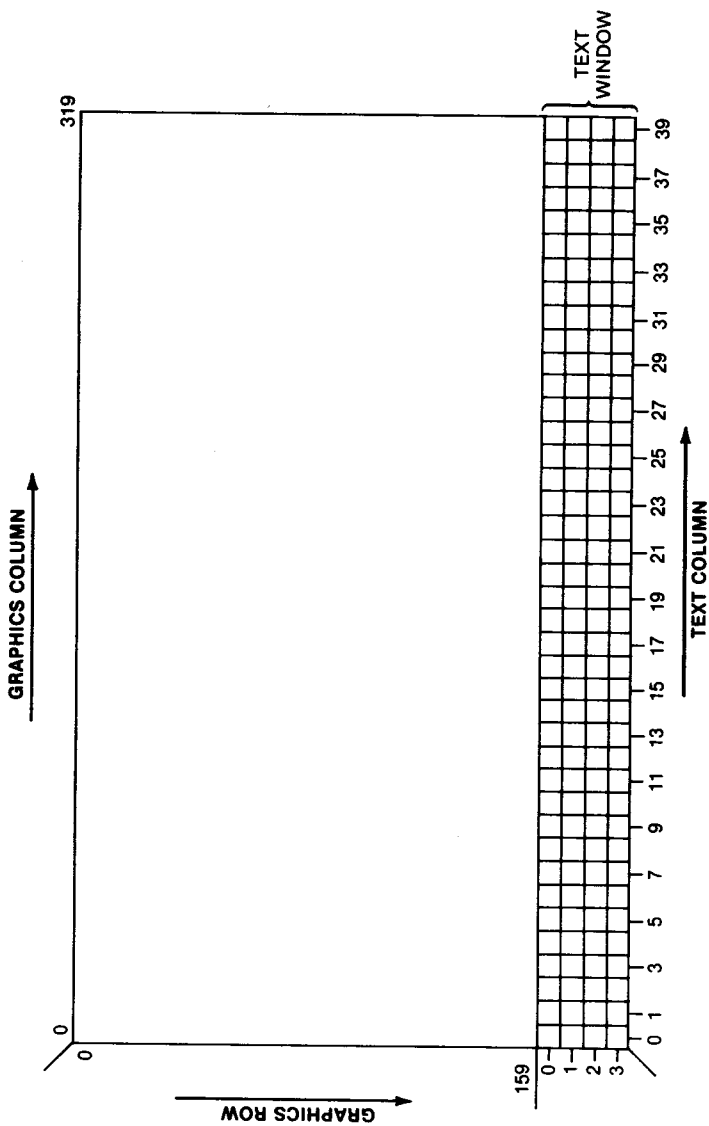


Fig. 4-40. Column/row format for the Mode-8 screen.

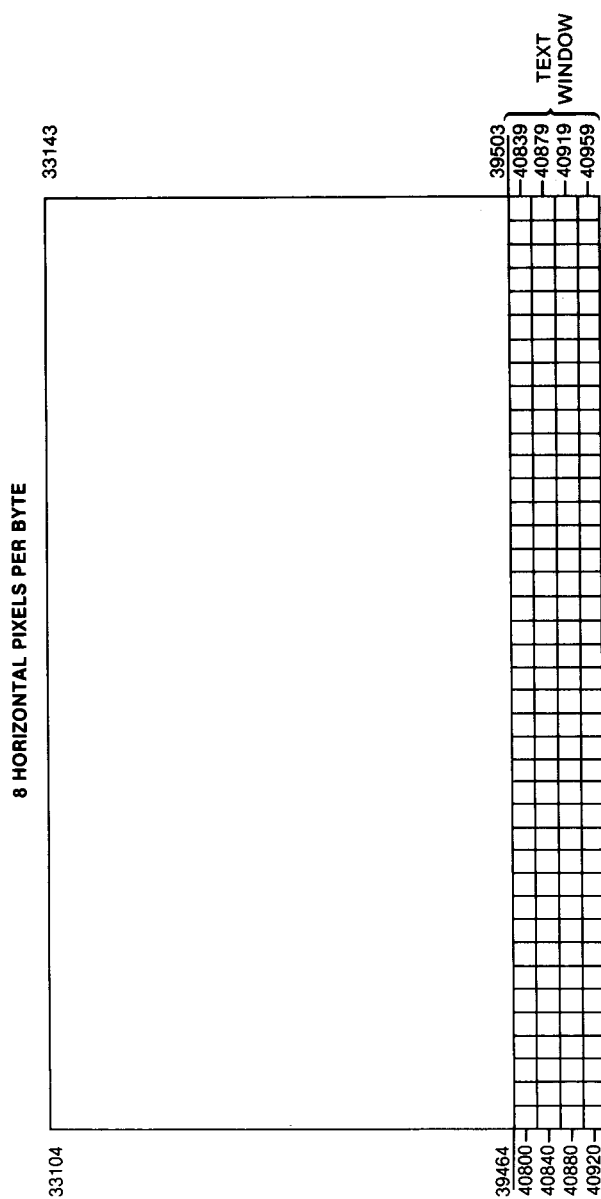


Fig. 4-41. Screen RAM address format for the Mode-8 screen. See a detailed listing of the addresses in Table 4-19.

**Table 4-19. Starting and Ending Addresses
for Each Row of the Mode-8 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 0	33104	33143	Row 42	34784	34823
Row 1	33144	33183	Row 43	34824	34863
Row 2	33184	33223	Row 44	34864	34903
Row 3	33224	33263	Row 45	34904	34943
Row 4	33264	33303	Row 46	34944	34983
Row 5	33304	33343	Row 47	34984	35023
Row 6	33344	33383	Row 48	35024	35063
Row 7	33384	33423	Row 49	35064	35103
Row 8	33424	33463	Row 50	35104	35143
Row 9	33464	33503	Row 51	35144	35183
Row 10	33504	33543	Row 52	35184	35223
Row 11	33544	33583	Row 53	35224	35263
Row 12	33584	33623	Row 54	35264	35303
Row 13	33624	33663	Row 55	35304	35343
Row 14	33664	33703	Row 56	35344	35383
Row 15	33704	33743	Row 57	35384	35423
Row 16	33744	33783	Row 58	35424	35463
Row 17	33784	33823	Row 59	35464	35503
Row 18	33824	33863	Row 60	35504	35543
Row 19	33864	33903	Row 61	35544	35583
Row 20	33904	33943	Row 62	35584	35623
Row 21	33944	33983	Row 63	35624	35663
Row 22	33984	34023	Row 64	35664	35703
Row 23	34024	34063	Row 65	35704	35743
Row 24	34064	34103	Row 66	35744	35783
Row 25	34104	34143	Row 67	35784	35823
Row 26	34144	34183	Row 68	35824	35863
Row 27	34184	34223	Row 69	35864	35903
Row 28	34224	34263	Row 70	35904	35943
Row 29	34264	34303	Row 71	35944	35983
Row 30	34304	34343	Row 72	35984	36023
Row 31	34344	34383	Row 73	36024	36063
Row 32	34384	34423	Row 74	36064	36103
Row 33	34424	34463	Row 75	36104	36143
Row 34	34464	34503	Row 76	36144	36183
Row 35	34504	34543	Row 77	36184	36223
Row 36	34544	34583	Row 78	36224	36263
Row 37	34584	34623	Row 79	36264	36303
Row 38	34624	34663	Row 80	36304	36343
Row 39	34664	34703	Row 81	36344	36383
Row 40	34704	34743	Row 82	36384	36423
Row 41	34744	34783	Row 83	36424	36463

**Table 4-19—cont. Starting and Ending Addresses
for Each Row of the Mode-8 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 84	36464	36503	Row 126	38144	38183
Row 85	36504	36543	Row 127	38184	38223
Row 86	36544	36583	Row 128	38224	38263
Row 87	36584	36623	Row 129	38264	38303
Row 88	36624	36663	Row 130	38304	38343
Row 89	36664	36703	Row 131	38344	38383
Row 90	36704	36743	Row 132	38384	38423
Row 91	36744	36783	Row 133	38424	38463
Row 92	36784	36823	Row 134	38464	38503
Row 93	36824	36863	Row 135	38504	38543
Row 94	36864	36903	Row 136	38544	38583
Row 95	36904	36943	Row 137	38584	38623
Row 96	36944	36983	Row 138	38624	38663
Row 97	36984	37023	Row 139	38664	38703
Row 98	37024	37063	Row 140	38704	38743
Row 99	37064	37103	Row 141	38744	38783
Row 100	37104	37143	Row 142	38784	38823
Row 101	37144	37183	Row 143	38824	38863
Row 102	37184	37223	Row 144	38864	38903
Row 103	37224	37263	Row 145	38904	38943
Row 104	37264	37303	Row 146	38944	38983
Row 105	37304	37343	Row 147	38984	39023
Row 106	37344	37383	Row 148	39024	39063
Row 107	37384	37423	Row 149	39064	39103
Row 108	37424	37463	Row 150	39104	39143
Row 109	37464	37503	Row 151	39144	39183
Row 110	37504	37543	Row 152	39184	39223
Row 111	37544	37583	Row 153	39224	39263
Row 112	37584	37623	Row 154	39264	39303
Row 113	37624	37663	Row 155	39304	39343
Row 114	37664	37703	Row 156	39344	39383
Row 115	37704	37743	Row 157	39384	39423
Row 116	37744	37783	Row 158	39424	39463
Row 117	37784	37823	Row 159	39464	39503
Row 118	37824	37863	Text window begins here		
Row 119	37864	37903			
Row 120	37904	37943	Row 0	40800	40839
Row 121	37944	37983	Row 1	40840	40879
Row 122	37984	38023	Row 2	40880	40919
Row 123	38024	38063	Row 3	40920	40959
Row 124	38064	38103			
Row 125	38104	38143			

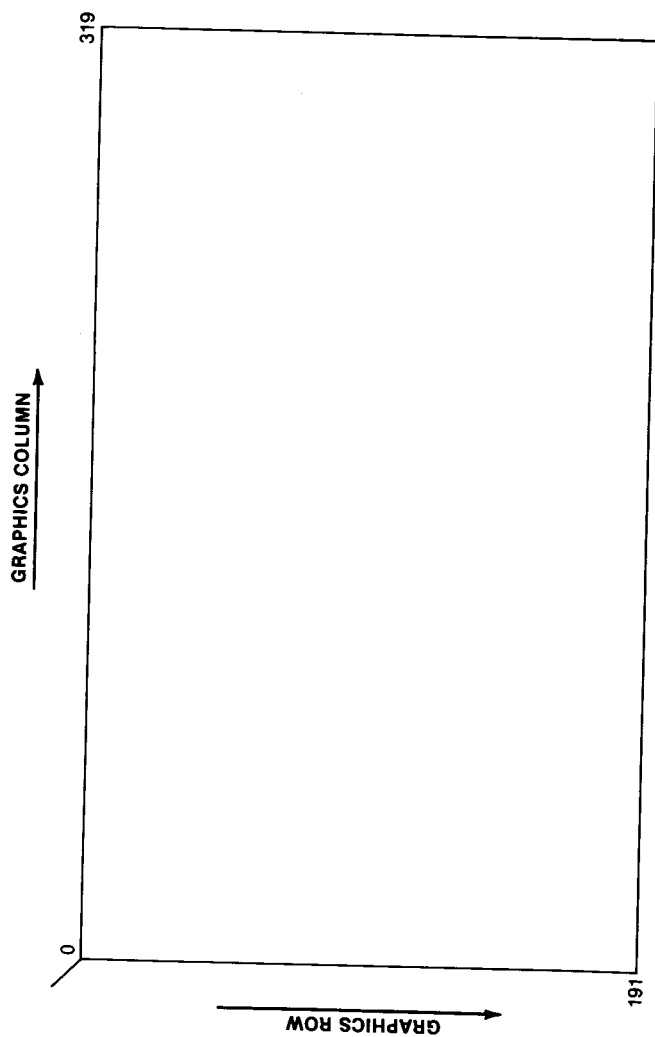


Fig. 4-42. Column/row format for the Mode-24 screen.

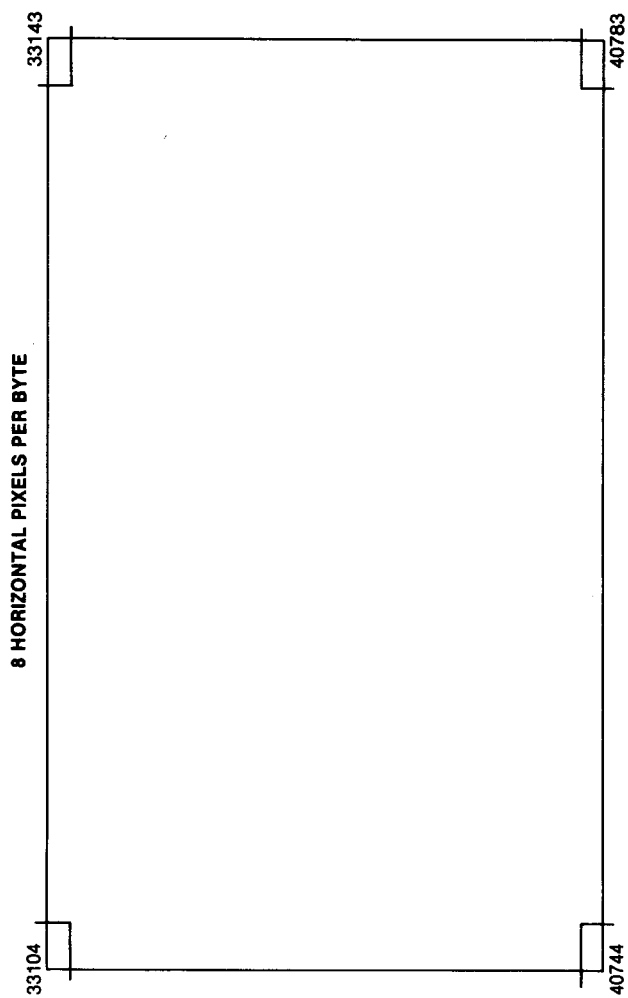


Fig. 4-43. Screen RAM address format for the Mode-24 screen. See a detailed listing of the addresses in Table 4-20.

**Table 4-20. Starting and Ending Addresses
for Each Row of the Mode-24 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 0	33104	33143	Row 36	34544	34583
Row 1	33144	33183	Row 37	34584	34623
Row 2	33184	33223	Row 38	34624	34663
Row 3	33224	33263	Row 39	34664	34703
Row 4	33264	33303	Row 40	34704	34743
Row 5	33304	33343	Row 41	34744	34783
Row 6	33344	33383	Row 42	34784	34823
Row 7	33384	33423	Row 43	34824	34863
Row 8	33424	33463	Row 44	34864	34903
Row 9	33464	33503	Row 45	34904	34943
Row 10	33504	33543	Row 46	34944	34983
Row 11	33544	33583	Row 47	34984	35023
Row 12	33584	33623	Row 48	35024	35063
Row 13	33624	33663	Row 49	35064	35103
Row 14	33664	33703	Row 50	35104	35143
Row 15	33704	33743	Row 51	35144	35183
Row 16	33744	33783	Row 52	35184	35223
Row 17	33784	33823	Row 53	35224	35263
Row 18	33824	33863	Row 54	35264	35303
Row 19	33864	33903	Row 55	35304	35343
Row 20	33904	33943	Row 56	35344	35383
Row 21	33944	33983	Row 57	35384	35423
Row 22	33984	34023	Row 58	35424	35463
Row 23	34024	34063	Row 59	35464	35503
Row 24	34064	34103	Row 60	35504	35543
Row 25	34104	34143	Row 61	35544	35583
Row 26	34144	34183	Row 62	35584	35623
Row 27	34184	34223	Row 63	35624	35663
Row 28	34224	34263	Row 64	35664	35703
Row 29	34264	34303	Row 65	35704	35743
Row 30	34304	34343	Row 66	35744	35783
Row 31	34344	34383	Row 67	35784	35823
Row 32	34384	34423	Row 68	35824	35863
Row 33	34424	34463	Row 69	35864	35903
Row 34	34464	34503	Row 70	35904	35943
Row 35	34504	34543	Row 71	35944	35983

**Table 4-20—cont. Starting and Ending Addresses
for Each Row of the Mode-24 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 72	35984	36023	Row 107	37384	37423
Row 73	36024	36063	Row 108	37424	37463
Row 74	36064	36103	Row 109	37464	37503
Row 75	36104	36143	Row 110	37504	37543
Row 76	36144	36183	Row 111	37544	37583
Row 77	36184	36223	Row 112	37584	37623
Row 78	36224	36263	Row 113	37624	37663
Row 79	36264	36303	Row 114	37664	37703
Row 80	36304	36343	Row 115	37704	37743
Row 81	36344	36383	Row 116	37744	37783
Row 82	36384	36423	Row 117	37784	37823
Row 83	36424	36463	Row 118	37824	37863
Row 84	36464	36503	Row 119	37864	37903
Row 85	36504	36543	Row 120	37904	37943
Row 86	36544	36583	Row 121	37944	37983
Row 87	36584	36623	Row 122	37984	38023
Row 88	36624	36663	Row 123	38024	38063
Row 89	36664	36703	Row 124	38064	38103
Row 90	36704	36743	Row 125	38104	38143
Row 91	36744	36783	Row 126	38144	38183
Row 92	36784	36823	Row 127	38184	38223
Row 93	36824	36863	Row 128	38224	38263
Row 94	36864	36903	Row 129	38264	38303
Row 95	36904	36943	Row 130	38304	38343
Row 96	36944	36983	Row 131	38344	38383
Row 97	36984	37023	Row 132	38384	38423
Row 98	37024	37063	Row 133	38424	38463
Row 99	37064	37103	Row 134	38464	38503
Row 100	37104	37143	Row 135	38504	38543
Row 101	37144	37183	Row 136	38544	38583
Row 102	37184	37223	Row 137	38584	38623
Row 103	37224	37263	Row 138	38624	38663
Row 104	37264	37303	Row 139	38664	38703
Row 105	37304	37343	Row 140	38704	38743
Row 106	37344	37383	Row 141	38744	38783

**Table 4-20—cont. Starting and Ending Addresses
for Each Row of the Mode-24 Screen RAM**

Row	Addresses		Row	Addresses	
	Start	End		Start	End
Row 142	38784	38823	Row 167	39784	39823
Row 143	38824	38863	Row 168	39824	39863
Row 144	38864	38903	Row 169	39864	39903
Row 145	38904	38943	Row 170	39904	39943
Row 146	38944	38983	Row 171	39944	39983
Row 147	38984	39023	Row 172	39984	40023
Row 148	39024	39063	Row 173	40024	40063
Row 149	39064	39103	Row 174	40064	40103
Row 150	39104	39143	Row 175	40104	40143
Row 151	39144	39183	Row 176	40144	40183
Row 152	39184	39223	Row 177	40184	40223
Row 153	39224	39263	Row 178	40224	40263
Row 154	39264	39303	Row 179	40264	40303
Row 155	39304	39343	Row 180	40304	40343
Row 156	39344	39383	Row 181	40344	40383
Row 157	39384	39423	Row 182	40384	40423
Row 158	39424	39463	Row 183	40424	40463
Row 159	39464	39503	Row 184	40464	40503
Row 160	39504	39543	Row 185	40504	40543
Row 161	39544	39583	Row 186	40544	40583
Row 162	39584	39623	Row 187	40584	40623
Row 163	39624	39663	Row 188	40624	40663
Row 164	39664	39703	Row 189	40664	40703
Row 165	39704	39743	Row 190	40704	40743
Row 166	39744	39783	Row 191	40744	40783

Chapter 5

Player/Missile Graphics

The ATARI home computer system features an enhanced animation package that generally goes by the name, *player/missile graphics*. The name is derived from the terminology that was applied to the original ATARI arcade game systems; and, indeed, the computer player/missile graphics retains many other general features of those highly successful systems.

Generally speaking, those games involved some figures that could be moved about on the screen (cannons, tanks, airplanes, and the like), and a different kind of figure—usually a simpler one—that is “fired” from the moving objects. The former was termed the *player* and the latter the *missile*. The terminology remains with ATARI systems, even though the original ideas have been superseded by a great many improvements and technological developments.

The ATARI 400/800 systems allow you to work with four different player and missile figures simultaneously (and even more if one cares to master some special programming tricks).

Player/missile graphics is something quite different from the other graphics and character modes that are described in the previous chapter. In the context of the discussions in this chapter, it is more meaningful to refer to those graphics and character modes as *playfield graphics*. The reasoning behind that bit of terminology is that the graphics modes provide colorful background and foreground material for the animated player-and-missile figures.

It is certainly possible to combine playfield and player/missile graphics. In fact, it is difficult to avoid the temptation of drawing some interesting and colorful playfield material to support the appearance and meaning of player/missile animation. The two kinds of ATARI graphics function quite differently from one another, however. And unlike the playfield graphics, there are no BASIC statements that relate directly to player/missile operations.

Anyone who does not feel comfortable with the notion of writing programs that are largely composed of PEEK and POKE statements will have some difficulty with player/missile graphics. All of the main player/missile operations refer to register addresses rather than convenient BASIC statements and functions.

The best approach to player/missile graphics is thus through the mechanisms of 6502 machine language.

In short, readers whose understanding of the ATARI programming is limited to BASIC are going to have some difficulty dealing with the full potential of player/missile graphics.

PLAYER AND MISSILE CONFIGURATIONS

The player and missile figures, up to four each, are all bit-mapped figures. Designing and loading the figures into a well-defined area of RAM is, in a sense, much the same as designing and entering custom text/graphics character sets as described at the end of Chapter 4.

Bit Maps for the Player Figures

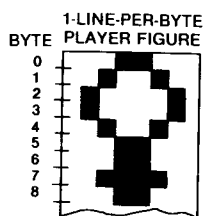
All player figures are mapped as exactly 8 bits wide. Exactly how those eight bits translate into width as the figures appear on the screen depends on the values that are POKEd into certain register locations. The important point here is that each line in a player figure is established by a single byte of data.

The number of successive 1-line bytes in the player-figure bit map determines its vertical length on the screen. The map must consist of exactly 128 or 256 bytes—no more, and no less. The 128-byte versions will use 1 byte of mapped data for every two horizontal scan lines on the tv or monitor; and the 256-byte version will use 1 byte for each horizontal scan line. The vertical resolution of the 128-byte version of a player figure is thus half that of the 256-byte version.

In either case, a player-figure bit map that uses meaningful color information throughout its entirety will produce a figure that extends the entire height of the screen. But that does not mean that every player figure has to appear that large. Portions of the bit map that contain binary 0 will be invisible; so you can create short figures by first setting all of the map addresses to 0, and then loading only those bytes that define the figure you want to see on the screen.

Designing a bit map for a player figure is a straightforward, if somewhat, tedious task. The usual technique is to score a sheet of graph paper such that you work with a figure window that is exactly 8 squares wide and as long as necessary (not exceeding the 128- or 255-bit limits).

Fig. 5-1 illustrates the map-development procedure for a 9-line player figure. The dark areas in Fig. 5-1A are to be colored, and the light areas are to be invisible. That pattern then translates to binary 1s and 0s as shown in Fig. 5-1B—1 = colored, 0 = invisible. The final step, shown in Fig. 5-1C, translates the binary codes to decimal and hexadecimal formats. (The decimal format is most useful when POKEing the data to the bit map, and the hexadecimal format is better when loading the bit map by means of a 6502 assembly language routine.)



(A) Sketch the figure on square graph paper, using 8 squares per line.

BINARY DATA

```

0 0 0 1 1 0 0 0
0 0 1 0 0 1 0 0
0 1 0 0 0 0 0 1 0
0 1 0 0 0 0 0 1 0
0 0 1 0 0 1 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 1 1 1 0 0 0
0 0 0 1 1 0 0 0

```

DECIMAL DATA

```

24
36
66
66
36
24
24
60
24

```

REMAINING 247 BYTES
ARE SET TO 0

(B) Translate the drawing into binary bytes: 0 = background color, 1 = figure color.

(C) Convert the binary format to decimal.

Fig. 5-1. Development of a player bit map.

Bearing in mind that the bit map for a player figure must be divided into 1-byte lines, a bit map merely suggests the general proportions of the player figure, and not necessarily its size as it will eventually appear on the screen. And the fact that this particular example uses only 9 bytes does not change the fact that its bit map must be either 128 or 256 bytes long. A later discussion suggests a simple technique for filling the remainder of a figure's bit map with zeros.

It is possible, but certainly not necessary, to develop three more player-figure bit maps. In fact, you can use the bit-map areas for any unused player figures for other purposes such as machine-language programs. Suppose, for example, that you are using player figures 0 and 1, and have no intention of using figures 2 and 3. You must commit the entire bit-map area for figures 0 and 1 to that purpose, but then the areas that are normally set aside for figures 2 and 3 are free for other uses.

Bit Maps for the Missile Figures

Missile figures, up to four of them, are likewise bit mapped into RAM. Mapping a 0 creates an "invisible" point and mapping a 1 creates a point of some desired color. And like player figures, missiles must be mapped as either 128 or 256 bytes—even if it means POKEing a lot of zeros to fill out that much space.

One big difference between the player and missile bit maps is the fact that each missile figure is defined in terms of 2 bits instead of 8. The missile map is 128 or 256 bytes long; but for a given missile figure, it is only 2 bits ($\frac{1}{4}$ byte) wide.

For the sake of RAM efficiency, the four missile figures are lined up side by side. Fig. 5-2 shows how a single byte accounts for line data for all four missile figures.

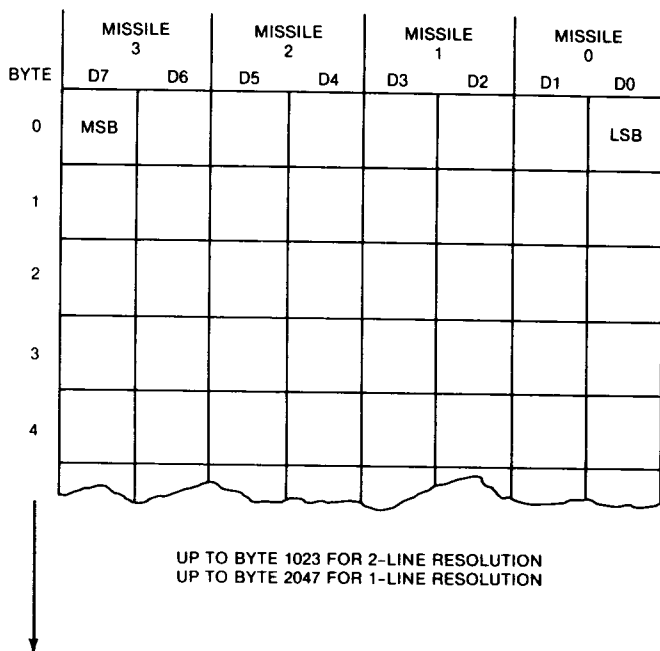


Fig. 5-2. Organization of the bit map for missile figures.

The two most-significant bits (D7 and D6) are devoted to missile figure M3, bits D5 and D4 are devoted to missile figure M2, bits D3 and D2 are devoted to missile figure M1, and the two least-significant bits (D1 and D0) are devoted to missile figure M0. Missile figure M0, for instance, will occupy the two least-significant bits of the missile bytes.

Having just 2 bits for each bit-mapped missile byte doesn't leave a whole lot of room for applying one's creative imagination. It is possible, however, to salvage the idea by combining two different missile figures into one (and making certain that they remain side by side throughout their animation routines).

The Overall Player/Missile Bit Map

Fig. 5-3 shows the two kinds of player/missile bit maps. You must use one or the other, and you must set aside the full amount of memory.

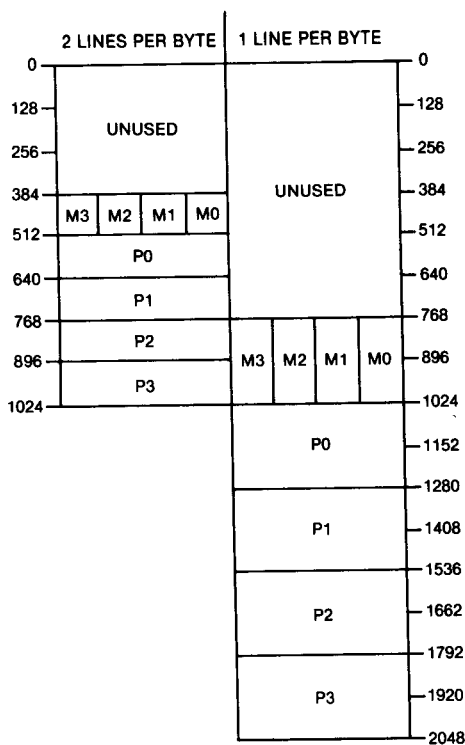


Fig. 5-3. The overall player/missile bit map for 2-line and 1-line vertical resolution.

The 2-lines-per-byte configuration devotes two horizontal scan lines per byte of player/missile data. In that instance, the space devoted to player and missile figures fills the screen—from top to bottom—with just 128 bytes per figure.

The 1-line-per-byte configuration offers a greater level of vertical resolution, but of course requires twice as much RAM to do the job. In that instance, there is 256 bytes set aside for each figure.

So if you elect to use the 2-lines-per-byte configuration, the player/missile bit map must occupy a total of 1024 bytes—no matter how many figures you choose to use, or how large or small they might be. Likewise, using the 1-line-per-byte configuration demands that you set aside 2048 bytes of memory for the bit maps.

The fact that you must set aside either 1024 or 2048 bytes of RAM for the player/missile bit map is not really a waste of good memory. If you are using just two player figures, for example, it is quite possible to use the RAM that is allocated for the other two player figures for machine-language programs. You are also free to use the *unused* RAM areas for your own purposes.

Actually, this business of having to fill the unused areas of the player/missile bit map with zeros is a rather trivial one. Once you have established the starting address of the bit map, called PMBASE, zeroing all the 2-lines-per-byte area is a matter of executing this sort of BASIC routine:

```
FOR N=0 TO 1023:POKE PMBASE+N,0:NEXT N
```

Or when using the 2048-byte version:

```
FOR N=0 TO 2047:POKE PMBASE+N,0:NEXT N
```

After setting *all* of the bit-map locations to zero, then the player/missile data and any custom machine-language programming can be POKEd or loaded into the map area. The only critical problem in the entire affair is that of determining the value of PMBASE—the base address for the player/missile bit map.

Setting the Starting Address of the Player/Missile Bit Map

The player/missile bit map must be located in a section of RAM where it can be protected from the operating system and BASIC, and yet in an area where there is no conflict with the playfield graphics operations. The usual location is just below the ANTIC display list for the playfield graphics; but even then it is necessary to contend with the fact that the location of the display list changes with the graphics modes and the amount of RAM installed in the computer (see Chapters 7 and 8).

What's more, PMBASE must begin at a 1k boundary value for a 1024-byte map, or at a 2k boundary value for a 2048-byte map.

Those might appear to be rather severe and troublesome restrictions upon the placement of the player/missile bit maps. But there is a reliable step-by-step procedure for carrying out the task.

The procedure begins by determining the starting addresses of all the playfield display lists you will be using, and then select the one having the lowest address.

You can determine the starting address of a display list by entering this program, setting up the playfield graphics mode, and then executing the program:

```
10 DISPL=PEEK(560)+256*(561)
20 GRAPHICS 0
30 PRINT DISPL
```

The LSB of the starting address of the display list is carried in RAM address 560, and the MSB is in address 561. The program thus PEEKs into those areas, converts the result to a decimal number and prints it to the Mode-0 screen.

The starting address of the player/missile bit map must then be at least 1024 or 2048 bytes lower—but at a 1k or 2k boundary value.

Table 5-1 shows all possible 1k and 2k boundary values for the ATARI system. If you are using the 1024-byte version of a player/missile bit map, subtract 1024 from the lowest display-list address, and then use the table to find the next-lower 1k boundary value. So if your lowest-numbered display list happens to begin at address 40540, and you are using a 1024-byte player/missile bit map, then:

$$40540 - 1024 = 39516$$

**Table 5-1. 1k and 2k
Boundary Values for RAM Addresses**

1k		2k
1024	32768	2048
2048	33792	4096
3072	34816	6144
4096	35840	8192
5120	36864	10240
6144	37888	12288
7168	38912	14336
8192	39936	16384
9216	40960	18432
10240	41984	20480
11264	43008	22528
12288	44032	24576
13312	45056	26624
14336	46080	28672
15360	47104	30720
16384	48128	32768
17408	49152	34816
18432	50176	36864
19456	51200	38912
20480	52224	40960
21504	53248	43008
22528	54272	45056
23552	55296	47104
24576	56320	49152
25600	57344	51200
26624	58368	53248
27648	59392	55296
28672	60416	57344
29696	61440	59392
30720	62464	61440
31744	63488	63488
	64512	

But 39516 does not represent a 1k boundary value. According to the 1k boundary-value list in Table 5-1, the next-lower address is 38912—and that is a 1k boundary value. Thus, the player/missile bit-map should begin at address 38912.

By way of another example, suppose that you want to use a 2048-byte map, and your lowest-addressed display list begins at 40266. Subtracting those values:

$$40266 - 2048 = 38218$$

And according to the 2k boundary table, the next-lower address is 36864. That is where the 2048-byte bit map should begin in this particular example.

If you do not feel inclined to do all of those calculations yourself, you can let the ATARI Home Computer do the work for you:

```
10 PRINT "SELECT ONE:"
20 PRINT "      1—SINGLE-LINE RESOLUTION
30 PRINT "      2—DOUBLE-LINE RESOLUTION
40 INPUT R
50 R=INT(R):IF NOT(R=1 OR R=2) THEN 40
60 DLST=PEEK(560)+256*PEEK(561)
70 IF R=1 THEN D=2048:GOTO 90
80 D=1024
90 PMBA=INT(DLST/D-1)*D
100 PRINT "DISPLAY LIST STARTS AT: ";DLST
110 PRINT "BIT MAP SHOULD START AT: ";PMB
```

Protecting the Player/Missile Bit Map

It is important to protect the player/missile bit map from operating-system and BASIC RAM operations. Once you know the starting address of the bit map, simply derive a 2-byte decimal version of it and POKE the LSB into RAM address 14 and the MSB into address 15. Those two addresses, often labeled APPEMHI set the highest RAM address that is available for operating-system and BASIC operations. A program that uses player/missile graphics should thus have POKEs to addresses 14 and 15 appearing

very early in the listing; certainly before attempting to set up the bit map.

ADJUSTING THE WIDTH OF THE PLAYER/MISSILE FIGURES

The selection of single- or double-line resolution has a lot to do with the vertical size of a player/missile figure; and, of course, so does the length of the bit map for those figures. The bit map, however, allows player figures to be only 8 pixels wide and missile figures to be just 1 pixel wide. Obviously, there are going to be instances where you want to use figures that are much wider than that—at least they should appear wider on the screen.

There are some registers available for expanding the width of the player/missile figures; specifically, it is possible to double or quadruple their width. Table 5-2 shows the RAM addresses of those width-control registers and the data bytes that should be POKEd to them in order to get 1x, 2x or 4x horizontal expansion.

Table 5-2. Player/Missile Figure Width Registers

Width-Register Label	Width-Register Address	Figure(s) Affected
SIZEP0	53256	Player 0
SIZEP1	53257	Player 1
SIZEP2	53258	Player 2
SIZEP3	53259	Player 3
SIZEM	53260	Missiles 0 through 3

POKE values:

0=normal width (1x bit-map width)

1=double width (2x bit-map width)

2=quadruple width (4x bit-map width)

Notice that it is possible to adjust the width of the four player figures independently. The scheme, however, allows just one width setting for all four missiles.

The default width is 1x. So if you plan to use alternative widths, it is important to POKE into the width-setting addresses very early in the program.

SETTING PLAYER/MISSILE COLORS

The color of the player/missile figures is set according to the same scheme that is used for playfield graphics. In that regard, Table 4-1 lists the basic hues and their data values, and Table 4-2 shows the relevant luminance values that must be summed with $16 \times \text{hue}$ to get a desired color. The player/missile figures, however, use a different set of RAM addresses for their color registers.

Table 5-3 shows the addresses of the color registers for player/missile figures 0 through 3. Notice that a given player figure and its corresponding missile share the same color register. Because of that particular feature, a player and its corresponding missile will always have the same color.

Table 5-3. Player/Missile Color Registers

Color-Register Label	Color-Register Address	Figures Affected
COLPM0	707	Player/Missile 0
COLPM1	708	Player/Missile 1
COLPM2	709	Player/Missile 2
COLPM3	710	Player/Missile 3

NOTE: See Tables 4-1 and 4-2 in Chapter 4 for listings of appropriate hue and luminance data values.

Naturally, it is important to POKE the desired player/missile colors into the color registers before initiating routines that display them on the screen.

INITIATING AND TERMINATING PLAYER/MISSILE GRAPHICS

The mere presence of a player/missile bit map and the setting of width color registers is not sufficient for initiating player/missile graphics modes. Two additional registers must be loaded with data that configures the entire scheme. The registers are GRCTL (address 53277) and DMACTL (address 559). GRCTL enables and disables the

player/missile graphics, and DMACTL is used for setting up a desired player/missile configuration.

Table 5-4 shows the relevant data values that can be POKEd to GRACTL.

**Table 5-4. POKE Operations for the GRACTL
(Graphics Control) Register**

POKEs to GRACTL	Function
POKE 53277,0	Disable all player/missile operations
POKE 53277,1	Enable missiles only
POKE 53277,2	Enable players only
POKE 53277,3	Enable both players and missiles

NOTE: Add 4 to the data values to latch all paddle trigger inputs. Using the values shown here will clear and disable the paddle trigger latches.

It is not sufficient to deal only with GRACTL. Every GRACTL should follow a POKE statement to DMACTL at address 559—a statement that actually configures the graphics operations.

Table 5-5 shows all possible playfield and player/missile configurations. Notice that the value POKEd to DMACTL must be greater than or equal to 32. POKEing values less than 32 will disable ANTIC altogether, thereby making it impossible to work with any kind of graphics—including normal Mode-0 text.

Those POKE-to-559 values offer a wide range of options. The default value, for example, is 34. From the table, you can see that it specifies a normal-sized playfield and disables the player/missile graphics.

It is often desirable to adjust those values during the execution of a program, but there is one parameter that ought to remain unchanged—the number of lines of resolution. That figure must match the choice you made earlier when setting up the player/missile bit map.

Generally, a programmer will first set up the DMACTL register, followed immediately by POKEing the appropriate value to GRACTL.

**Table 5-5. POKE Values for Configuring the
Player/ Missile Operations at the DMACTL
(Direct-Memory-Address Control) Register**

POKE Value	Playfield Configuration	Missile Status	Player Status	Vertical Resolution
32	none	disable	disable	not relevant
33	narrow	disable	disable	not relevant
34	standard	disable	disable	not relevant
35	wide	disable	disable	not relevant
36	none	enable	disable	2-line
37	narrow	enable	disable	2-line
38	standard	enable	disable	2-line
39	wide	enable	disable	2-line
40	none	disable	enable	2-line
41	narrow	disable	enable	2-line
42	standard	disable	enable	2-line
43	wide	disable	enable	2-line
44	none	enable	enable	2-line
45	narrow	enable	enable	2-line
46	standard	enable	enable	2-line
47	wide	enable	enable	2-line
48	none	disable	disable	not relevant
49	narrow	disable	disable	not relevant
50	standard	disable	disable	not relevant
51	wide	disable	disable	not relevant
52	none	enable	disable	1-line
53	narrow	enable	disable	1-line
54	standard	enable	disable	1-line
55	wide	enable	disable	1-line
56	none	disable	enable	1-line
57	narrow	disable	enable	1-line
58	standard	disable	enable	1-line
59	wide	disable	enable	1-line
60	none	enable	enable	1-line
61	narrow	enable	enable	1-line
62	standard	enable	enable	1-line
63	wide	enable	enable	1-line

MOVING THE PLAYER/MISSILE FIGURES

Moving player and missile figures horizontally across the screen is a relatively simple operation—just POKE the desired column number into a register that is designated for that purpose. The POKEd value will determine the column location of the left-hand side of the bit-mapped figure, and the range of useful values depends on the figure's bit-mapped width and its width-register setting.

Table 5-6 lists the horizontal position registers for the eight player and missile figures.

**Table 5-6. The Horizontal 6-Position Registers
for the Player/Missile Figures**

Label	Address	Figure Affected
HPOSPO	53248	Player 0
HPOSP1	53249	Player 1
HPOSP2	53250	Player 2
HPOSP3	53251	Player 3
HPOSM0	53252	Missile 0
HPOSM1	53253	Missile 1
HPOSM2	53254	Missile 2
HPOSM3	53255	Missile 3

POKEing horizontal-position values less than 40 will generally bury the figure within the horizontal blanking area of the raster scan. That renders the figure virtually invisible—which isn't necessarily an undesirable situation. It is far easier to remove a figure from the screen by POKEing 0 into its horizontal-position register than it is to hide it by any other means.

Vertical motion is generated by actually moving the figure through its own bit map. The bit map for a 2-line-per-byte player figure is 128 bytes long. As described earlier, that much RAM must be allocated for every player figure. Few player figures will be that long, however, and the space that remains is space that can be used for moving the figure in the vertical directions.

That earlier discussion also implied that the top of a figure should begin at the start of its bit-map area. That isn't necessarily the case; but when a figure is located at the top of its bit-map area, it will appear at the top of the screen. Downward vertical motion is then achieved by pushing the figure deeper into its bit-map area (into successively higher RAM addresses).

Fig. 5-3 shows the memory maps for the overall player/missile bit map. If you are using a 2-line resolution, the bit map for player figure 0 begins at a known address: the starting address of the bit map (PMBASE) + 512. It is then possible to move the figure downward by means of a routine that copies the bit map with successively higher addresses.

That sort of vertical-motion operation can be rather time consuming when run in BASIC, however, so it is generally considered a good practice to do the job with the help of a simple machine-language move operation.

PLAYER/PLAYFIELD PRIORITIES

ATARI computers, with their special ANTIC graphics devices, make it rather easy to create the impression that one object is moving behind or in front of another. There is a priority-select register that lets you determine whether a given figure has priority over, or will move in front of, another figure on the screen.

That register is normally labeled GPRIOR, and it is located at address 632. Table 5-7 shows the values that should be POKEd to GPRIOR in order to establish the orders of priority.

Executing a POKE 632,1 will give the player/missile figures priority over any playfield objects, and any playfield objects will take priority over the background color.

**Table 5-7. POKE Values for GPRIOR at Address 632.
These Values Set the Priority of the Player, Missile, and
Playfield Figures**

Value	Priorities
1	Player/missiles 0, 1, 2, and 3 Playfield registers 0, 1, 2, and 3 Background
2	Player/missiles 0 and 1 Playfield registers 0, 1, 2, and 3 Player/missiles 2 and 3
4	Playfield registers 0, 1, 2, and 3 Player/missiles 0, 1, 2, and 3 Background
8	Playfield registers 0 and 1 Player/missiles 0, 1, 2, and 3 Playfield registers 2 and 3 Background

Add 16—All four missiles take on the color specified at playfield register 3; a condition often regarded as one that adds a fifth player to the screen.

Add 32—Overlapping of player/missiles 0 and 1, or 2 and 3, will create a third color in the overlapped region. Other combinations of overlapped figures will always show black in the overlapped region.

On the other hand, executing a POKE 632,8 will give any playfield graphic that is plotted from color registers 0 and 1 priority over all of the player/missile figures; but then all player/missile figures will take priority over any playfield graphic that is plotted from color registers 2 or 3. The background, as usual, takes the lowest precedence.

It is important to recall that only the 4-color graphics modes (Modes 3, 5, and 7) use all four playfield color registers. Using the 2-color graphics modes limits the capability of this powerful animation and graphics tool.

COLLISION DETECTION

Yet another feature of the ATARI graphics system is a built-in scheme for detecting a collision between figures. The scheme not only detects a collision, but returns values that indicate the kinds of figures involved.

There are 16 collision-detection registers and an additional register that is used for clearing them. Normally, a program will clear the collision-detection registers and then poll them from time to time to see whether or not certain collision events have taken place. After some relevant collisions have occurred, the program can take some appropriate action and clear the registers once again.

The collision-status clearing register, HITCLR, is located at address 53278; and POKEing any value greater than zero will cause the system to clear the collision-detection registers.

Table 5-8 summarizes the collision format for collisions

Table 5-8. Values Returned by PEEKing Into the Missile-to-Playfields Collision Registers

<i>n</i>	Collision Scenario
0	no collision since most recent HITCLR; POKE 53278,255
1	Playfield from color register 0
2	Playfield from color register 1
3	Playfields from color registers 0 and 1
4	Playfield from color register 2
5	Playfields from color registers 0 and 2
6	Playfields from color registers 1 and 2
7	Playfields from color registers 0, 1 and 2
8	Playfield from color register 3
9	Playfields from color registers 0 and 3
10	Playfields from color registers 1 and 3
11	Playfields from color registers 0, 1, and 3
12	Playfields from color registers 2 and 3
13	Playfields from color registers 0, 2, and 3
14	Playfields from color registers 1, 2, and 3
15	Playfields from color registers 0, 1, 2, and 3

*n*PEEK(53248) for Missile-0 Collisions with Playfields

*n*PEEK(53249) for Missile-1 Collisions with Playfields

*n*PEEK(53250) for Missile-2 Collisions with Playfields

*n*PEEK(53251) for Missile-3 Collisions with Playfields

between the missile figures and any playfield objects that are plotted from color registers 0 through 3. Whenever it is necessary to see whether or not a collision has occurred between missile figure 2 and a playfield object drawn from color register 1, this sort of BASIC statement is in order:

IF PEEK 53250=2 THEN PRINT "BOOM"

Table 5-9 summarizes the same sort of information; but in this case, the collision-detection operations refer to collisions between player figures and playfield colors.

Table 5-9. Values Returned by PEEKing Into the Player-to-Playfields Collision Registers

<i>n</i>	Collision Scenario
0	no collision since most recent HITCLR; POKE 53278,255
1	Playfield from color register 0
2	Playfield from color register 1
3	Playfields from color registers 0 and 1
4	Playfield from color register 2
5	Playfields from color registers 0 and 2
6	Playfields from color registers 1 and 2
7	Playfields from color registers 0, 1, and 2
8	Playfield from color register 3
9	Playfields from color registers 0 and 3
10	Playfields from color registers 1 and 3
11	Playfields from color registers 0, 1, and 3
12	Playfields from color registers 2 and 3
13	Playfields from color registers 0, 2, and 3
14	Playfields from color registers 1, 2, and 3
15	Playfields from color registers 0, 1, 2, and 3

*n*PEEK(53252) for Player-0 Collisions with Playfields

*n*PEEK(53253) for Player-1 Collisions with Playfields

*n*PEEK(53254) for Player-2 Collisions with Playfields

*n*PEEK(53255) for Player-3 Collisions with Playfields

Again, the same general format appears in Table 5-10. The collisions in this instance, however, are between missiles and players.

Tables 5-11 through 5-14 show the PEEK values that are returned from registers that detect collisions between two different player figures. The summary is divided into four separate parts because a collision between a player figure and itself is not a relevant, or even meaningful, situation.

Table 5-10. Values Returned by PEEKing Into the Missile-to-Players Collision Registers

<i>n</i>	Collision Scenario
0	no collision since most recent HITCLR; POKE 53278,255
1	Player 0
2	Player 1
3	Players 0 and 1
4	Player 2
5	Players 0 and 2
6	Players 1 and 2
7	Players 0, 1, and 2
8	Player 3
9	Players 0 and 3
10	Players 1 and 3
11	Players 0, 1, and 3
12	Players 2 and 3
13	Players 0, 2, and 3
14	Players 1, 2, and 3
15	Players 0, 1, 2, and 3

n=PEEK(53256) for Missile-0 Collisions with Players

n=PEEK(53257) for Missile-1 Collisions with Players

n=PEEK(53258) for Missile-2 Collisions with Players

n=PEEK(53259) for Missile-3 Collisions with Players

Table 5-11. Values Returned by PEEKing Into the Player-to-Players Collision Registers (Player 0 to Players 1, 2, and 3)

<i>n</i>	Collision Scenario
0	no collision since most recent HITCLR; POKE 53278,255
2	Player 1
3	Player 1
4	Player 2
5	Player 2
6	Players 1 and 2
7	Players 1 and 2
8	Player 3
9	Player 3
10	Players 1 and 3
11	Players 1 and 3
12	Players 2 and 3
13	Players 2 and 3
14	Players 1, 2, and 3
15	Players 1, 2, and 3

*n*PEEK(53260) for Player-0 Collisions with Players 1, 2, and 3

**Table 5-12. Values Returned by PEEKing Into the
Player-to-Players Collision Registers
(Player 1 to Players 0, 2, and 3)**

<i>n</i>	Collision Scenario
0	no collision since most recent HITCLR; POKE 53278,255
1	Player 0
3	Player 0
4	Player 2
5	Players 0 and 2
6	Player 2
7	Players 0 and 2
8	Player 3
9	Players 0 and 3
10	Player 3
11	Players 0 and 3
12	Players 2 and 3
13	Players 0, 2, and 3
14	Players 2 and 3
15	Players 0, 2, and 3

n=PEEK(53261) for Player-1 Collisions with Players 0, 2, and 3

**Table 5-13. Values Returned by PEEKing Into the
Player-to-Players Collision Registers
(Player 2 to Players 0, 1, and 3)**

<i>n</i>	Collision Scenario
0	no collision since most recent HITCLR; POKE 53278,255
1	Player 0
2	Player 1
3	Players 0 and 1
5	Player 0
6	Player 1
7	Players 0 and 1
8	Player 3
9	Players 0 and 3
10	Players 1 and 3
11	Players 0, 1, and 3
12	Player 3
13	Players 0 and 3
14	Players 1 and 3
15	Players 0, 1, and 3

n=PEEK(53262) for Player-2 Collisions with Players 0, 1, and 3

**Table 5-14. Values Returned by PEEKing Into the
Player-to-Players Collision Registers
(Player 3 to Players 0, 1, and 2)**

<i>n</i>	Collision Scenario
0	no collision since most recent HITCLR; POKE 53278,255
1	Player 0
2	Player 1
3	Players 0 and 1
4	Player 2
5	Players 0 and 2
6	Players 1 and 2
7	Players 0, 1, and 2
9	Player 0
10	Player 1
11	Players 0 and 1
12	Player 2
13	Players 0 and 2
14	Players 1 and 2
15	Players 0, 1, and 2

*n*PEEK(53264) for Player-3 Collisions with Players 0, 1, and 2

Chapter 6

More About I/O Operations

A computer is useless without being able to perform operations that let it accept data from other devices, including the keyboard, and let it direct information to other devices, including the display screen. Computer users often take such operations for granted, but the ATARI Operating System offers some fine opportunities for redirecting the normal flow of information and, indeed, creating new ones.

WORKING WITH THE PROGRAM RECORDER I/O

Chapter 1 describes the most commonly used techniques for saving and loading tokenized BASIC programs at the program recorder. The general idea is to execute CSAVE and CLOAD commands or, alternatively, SAVE "C:" and LOAD "C:" commands.

The following discussions extend the range of techniques to include ATASCII-coded BASIC and pure data files.

ATASCII-Coded BASIC Programs—LIST“C:” and ENTER“C:”

The mechanical operations involved in saving and loading BASIC programs that are saved in an ATASCII, or nontokenized, format are essentially the same as those already described for tokenized programs. The only real difference is the nature of the commands.

To save an ATASCII-coded version of a BASIC program on the program recorder, find the desired place on the cassette tape and note the tape-counter reading.

Set the recorder to its record mode and enter this command:

LIST “C:”

That will list the entire BASIC program, in the ATASCII-coded format, to the program recorder. You know that the operation is finished when the BASIC prompt symbol and READY message reappear on the screen. Turn off the program recorder at that time.

Alternatively, you can record selected portions of a BASIC program that is resident in the system. This procedure is especially helpful whenever you want to save some commonly used subroutines for future programs. Whereas the LIST “C:” command will save the entire program on cassette tape, this command will list a portion of it:

LIST “C:”, *strtline*, *lastline*

where *strtline* is the line number of the first line to be recorded, and *lastline* is the final line number in the segment.

The LIST command thus saves an ATASCII-coded version of a BASIC program, or a selected portion of it, onto cassette tape. The ENTER command is used for loading that material back into RAM from the program recorder.

The mechanical procedures for using the ENTER command are quite similar to those for using the CLOAD command. The ENTER command, however, must be used with programs that were originally saved under the LIST command.

First, enter this command at the ATARI console:

ENTER "C:"

The system will respond to that command by beeping the console loudspeaker twice. That is the time to cue the tape to the beginning of the ATASCII-coded program you want to load. After doing that, depress the PLAY lever on the program recorder and strike any key on the ATARI console (except the BREAK key). That will begin the loading operation. The loading session is done when the BASIC cursor and READY message reappear on the screen.

The important advantage of the LIST/ENTER operations is that they permit the merging of smaller program segments into a larger program in RAM. Recall that the LIST program lets you specify selected portions of a program to be saved on cassette tape. That is one difference. The big difference, however, is that the ENTER command, unlike its CLOAD counterpart, does not erase existing BASIC programming from the system RAM; rather it adds the previously saved routines to the existing program.

In the event that there are duplicate line numbers, the most recently ENTERed segment will take precedence; in effect, writing over lines of programming having the same line numbers. When contemplating the use of these merging operations, it is thus very important to make sure that the LISTed program line numbers will not duplicate important program lines when you return them to the system via the ENTER command.

Given some practice and some careful planning, it is quite possible to use the LIST/ENTER commands to build very large programs in a highly efficient fashion from routines that you developed and LISTed to the program recorder at previous times.

The LIST and ENTER commands that refer to the program recorder automatically select IOCB Channel 7. That channel must not be open for any other purpose at the time you execute either command.

Nonprogram Files—PRINT and INPUT

In the context of saving and loading information to the program recorder, the PRINT and INPUT statements are useful for working with files that are composed of information other than lines of BASIC programming. Unlike the program saving and loading operations, PRINT and INPUT must be included in a program of their own—the scheme cannot work directly from the keyboard.

The following example suggests a programming procedure for a relatively simple data-recording operation:

```
10 OPEN #7,8,0,"C:"  
20 FOR N=0 TO 1000  
30 PRINT #7,N;  
40 NEXT N
```

Line 10 opens IOCB Channel 7 for output to the program recorder, the FOR . . . NEXT loop generates the numerical values for variable N, and line 30 outputs those values to the program recorder.

Upon executing the OPEN statement in line 10, the system will beep the console loudspeaker twice to signal the time to set the program recorder to its RECORD mode and strike any key (except the BREAK key) to begin the recording operation.

The PRINT statement loads the values to the tape in groups of 128; and you will notice a pause between each group. Turn off the program recorder when the program ends.

The example illustrates the fact that it is important to open an IOCB channel prior to executing the PRINT command. In this context, the general form of that OPEN statement is

OPEN #*chan*,8,*gap*

where *chan* is the desired IOCB channel for the operation, and *gap* is a value of zero or 128. Setting *gap* to zero causes the program recorder to pause while the system is loading the next 128 bytes of data to the output buffer. Setting *gap* shortens that interval, but risks inserting garbage between successive segments of data on the cassette tape.

The PRINT statement must specify the same IOCB channel that is opened by the previous OPEN statement:

PRINT #*chan*, . . .

The following example suggests a way to reload the numerals that were saved by the previous example of the PRINT statement:

```
10 DIM X(1000)
20 OPEN #7,4,0,"C:"
30 FOR N=0 TO 1000
40 GET #7,X(N) INPUT #7,X(N)
50 NEXT N
```

Line 10 dimensions subscripted variable X, and line 20 opens IOCB Channel 7 for input from the program recorder, using normal inter-record delays. Lines 30 through 50 then read the data from the cassette tape, assigning each numerical value to the subscripted variable, X.

Upon executing the OPEN statement, the system will beep the console loudspeaker twice, thus signaling the time to set the program recorder to the PLAY mode and strike any key (except BREAK) to continue.

The PRINT and INPUT techniques work equally well with string data and, indeed, combinations of numerical and string data. It is absolutely necessary, however, to make certain that the variable names are organized such that there is no type mismatch anywhere through the saving and loading routines.

Nonprogram Files—PUT and GET

BASIC's PUT and GET statements offer an alternative means for working with file data. Unlike PRINT and INPUT, however, PUT and GET operations are limited to numerical values (although strings can be converted to ATASCII codes before they are PUT to the program recorder. But generally speaking, the PUT/GET routines are quite similar to the PRINT/INPUT routines just described.

The following example suggests a programming procedure for a relatively simple data-recording operation:

```
10 OPEN #7,8,0,"C:"  
20 FOR N=0 TO 1000  
30 PUT #7,N  
40 NEXT N
```

Line 10 opens IOCB Channel 7 for output to the program recorder, the FOR . . . NEXT loop generates the numerical values for variable N, and line 30 outputs those values to the program recorder.

Upon executing the OPEN statement in line 10, the system will beep the console loudspeaker twice to signal the time to set the program recorder to its RECORD mode and strike any key (except the BREAK key) to begin the recording operation.

The PUT statement loads the values to the tape in groups of 128; and you will notice a pause between each group. Turn off the program recorder when the program ends.

As with the PRINT-recording technique, an IOCB channel must be opened for output to the program recorder before the PUT operations begins:

OPEN #*chan*,8,*gap*

where *chan* is the desire IOCB channel for the operation, and *gap* is a value of zero or 128 for the purpose described for the PRINT/INPUT operations.

The subsequent PUT statement must specify the same IOCB channel that is opened by the OPEN statement:

PUT #*chan*,*x*

where *chan* is the IOCB channel number and *x* is any numeric variable, constant or expression.

The following example suggests a way to reload the numerals that were saved by the previous example of the PUT statement:

```
10 DIM X(1000)
20 OPEN #7,4,0,"C:"
30 FOR N=0 TO 1000
40 GET #7,X(N)
50 NEXT N
```

Line 10 dimensions subscripted variable X, and line 20 opens IOCB Channel 7 for input from the program recorder, using normal inter-record delays. Lines 30 through 50 then read the data from the cassette tape, assigning each numerical value to the subscripted variable, X.

Upon executing the OPEN statement, the system will beep the console loudspeaker twice, thus signaling the time to set the program recorder to the PLAY mode and strike any key (except BREAK) to continue.

It is possible to use the PUT/GET operations with string data if the strings are first converted to their ATASCII codes.

WORKING WITH THE DISK DRIVE I/O

Chapter 1 describes the most commonly used techniques for saving and loading tokenized BASIC programs on a diskette by means of SAVE "D:" and LOAD "D:" commands, respectively.

The following discussions extend the range of techniques to include ATASCII-coded BASIC and pure data files.

ATASCII-Coded BASIC Programs—LIST "D:" and ENTER "D:"

The mechanical operations involved in saving and loading BASIC programs that are saved in an ATASCII, or nontokenized, format are essentially the same as those described for tokenized programs. The only real difference is the nature of the commands.

Use a command of this form to save an ATASCII-coded version of a BASIC program on a selected disk drive system:

LIST "D[n]:*filename* [*ext*]"

That will list the entire BASIC program to disk-drive *n* under the name, *filename*, and with an optional extension, *ext*. See the previous discussion for general rules regarding the selection of filenames and extensions.

Example:

LIST "D:SILLY.FUN"

That LIST command will use the default disk drive (usually drive #1) for saving an entire BASIC program—SILLY.FUN—in an ATASCII-code format.

Alternatively, you can record selected portions of a BASIC program that is resident in the system. This procedure is especially helpful when you want to save some commonly used subroutines for future programs. Whereas the previous saves the entire program, this form of the command saves a selected portion of it:

```
LIST"D[n]:filename[.ext]",strtline,lastline
```

where *strtline* is the line number of the first line to be recorded, and *lastline* is the final line number in the segment.

Example:

```
LIST"D:MIXER",200,300
```

That command will save the BASIC programming between lines 200 and 300, inclusively, on the default disk drive under the name MIXER.

The ENTER command is used for loading ATASCII-coded BASIC programs that were originally saved with the LIST command. The general form of the command is;

```
ENTER"D[n]:filename[.ext]"
```

The important advantage of the LIST/ENTER operations is that they permit the merging of smaller program segments into a larger program in RAM. Recall that the LIST program lets you specify selected portions of a program to be saved on a diskette. That is one difference. The big difference, however, is that the ENTER command, unlike its SAVE counterpart, does not erase existing BASIC programming from the system RAM; rather it adds the previously saved routines to the existing program.

In the event that there are duplicate line numbers, the most recently ENTERed segment will take precedence; in effect, writing over lines of programming having the same line numbers. When contemplating the use of these merging operations, it is thus very important to make sure that the LISTed program line numbers will not duplicate important program lines when you return them to the system via the ENTER command.

Given some practice and some careful planning, it is quite possible to use the LIST/ENTER commands to build very large programs in a highly efficient fashion from routines that you developed and LISTed on a diskette at previous times.

Nonprogram Files—PRINT and INPUT

In the context of saving and loading information to a selected disk drive, the PRINT and INPUT statements are useful for working with files that are composed of information other than lines of BASIC programming. Unlike the operations for saving and loading BASIC programs, PRINT and INPUT must be included in a program of their own—the scheme cannot work directly from the keyboard.

The following example suggests a programming procedure for a relatively simple data-recording operation:

```
10 OPEN #7,8,0,"D:NUMBERS.FIL"  
20 FOR N=0 TO 1000  
30 PRINT #7,N;  
40 NEXT N
```

Line 10 opens IOCB Channel 7 for writing a new program to the default disk drive under the name NUMBERS.FIL. The FOR . . . NEXT loop generates the numerical values for variable N, and line 30 outputs those values to the program recorder.

Upon executing the OPEN statement in line 10, the disk drive will begin running, and it will continue running until the saving routine is completed.

The example illustrates the fact that it is important to open an IOCB channel prior to executing the PRINT command. In this context, the general form of the OPEN statement is

OPEN #*chan*,*task*,0,"D:*filename*"

where *chan* is the desired IOCB channel for the operation and *task* defines the exact nature of the disk operation. It is possible to specify an alternate disk drive, *Dn*, and assign an extension to the filename. There are two different task assignments that are useful for PRINT operations:

Write a new file—8

Append an existing file—9

The example suggests a *task* value of 8 so that NUMBERS.FIL will be opened as a new file. You can add data to an existing file, though, by using a *task* value of 9.

The PRINT statement must specify the same IOCB channel that is opened by the previous OPEN statement:

PRINT #*chan*, . . .

The following example suggests a way to reload the numerals that were saved by the previous example of the PRINT statement:

```
10 DIM X(1000)
20 OPEN #7,4,0,"D:NUMBERS.FIL"
30 FOR N=0 TO 1000
40 GET #7,X(N)
50 NEXT N
```

Line 10 dimensions subscripted variable X, and line 20 opens IOCB Channel 7 for reading data from a disk file named NUMBERS.FIL. Lines 30 through 50 then read the data from that file, assigning each numerical value to the subscripted variable, X.

The PRINT and INPUT techniques work equally well with string data and, indeed, combinations of numerical and string data. It is absolutely necessary, however, to make certain that the variable names are organized such that there is no type mismatch anywhere through the saving and loading routines.

Nonprogram Files—PUT and GET

BASIC's PUT and GET statements offer an alternative means for working with file data. Unlike PRINT and INPUT, however, PUT and GET operations are limited to numerical values (although strings can be converted to ATASCII codes before they are PUT to the program recorder). But generally speaking, the PUT/GET routines are quite similar to the PRINT/INPUT routines just described.

The following example suggests a programming procedure for a relatively simple data-recording operation:

```
10 OPEN #7,8,0,"D:NUMBERS.FIL"  
20 FOR N=0 TO 1000  
30 PUT #7,N  
40 NEXT N
```

Line 10 opens IOCB Channel 7 for writing a new file to the default disk drive. The FOR . . . NEXT loop generates the numerical values for variable N, and line 30 outputs those values to the disk.

As with the PRINT-recording technique, an IOCB channel must be opened for output to the program recorder before the PUT operation begins. The PUT statement must then specify the same IOCB channel number. The general form of PUT statement is:

PUT #*chan*,*x*

where *chan* is the IOCB channel number and *x* is any numeric variable, constant, or expression.

The following example suggests a way to reload the numerals that were saved by the previous example of the PUT statement:

```
10 DIM X(1000)
20 OPEN #7,4,0,"D:NUMBERS.FIL"
30 FOR N=0 TO 1000
40 GET #7,X(N)
50 NEXT N
```

Line 10 dimensions subscripted variable X, and line 20 opens IOCB Channel 7 for reading data from a disk file named NUMBERS.FIL. Lines 30 through 50 then read the data from the cassette tape, assigning each numerical value to the subscripted variable, X.

It is possible to use the PUT/GET operations with string data if the strings are first converted to their ATASCII codes with the help of the ASC or ADR functions.

SAVING, LOADING, AND RUNNING BINARY FILES UNDER DOS

Chapter 1 describes most of the DOS operations that can be conducted directly from the DOS menu. The idea is to bring up the menu by entering the DOS command from BASIC, select one of the menu options, and then follow the prompting messages from there. Returning to BASIC is a matter of selecting DOS menu option B (assuming that the BASIC cartridge is installed) or by striking the SYSTEM RESET key.

It was not appropriate to describe the three DOS menu options that deal with binary, or machine-language, files. This is a better place to deal with that matter.

Saving Binary Programs and Data

The DOS menus for both versions, 1.0 and 2.0S, show BINARY SAVE as option K. They both accomplish the same task—saving specified block of binary-coded information on disk—but the mechanical procedures are somewhat different.

Under DOS version 1.0, selecting the K option brings up this prompting message:

SAVE-GIVE FILE,START,END

The system is expecting you to enter, in turn, a filename, the start address of the data to be saved, and the last address. Both addresses must be entered in a hexadecimal format.

When the data is loaded back into the computer at some later time, you can have things arranged so that it will begin execution immediately. The procedure requires several steps, but the results can be quite satisfying. Before executing the DOS command to see the menu, POKE a 2-byte binary version of the starting address of the machine-language routine into RAM addresses 736 and 737—LSB followed by MSB. Then execute the DOS command, and select menu item K. Respond to the prompting message as before, but this time append the filename with /A. The latter step, along with setting the starting address into locations 736 and 737, will save the starting address on the disk as well as the machine-language routine, itself.

Having done that, electing menu option L will not only load the program, but begin running it immediately.

Matters are somewhat simple when saving binary files under DOS 2.0S. Upon selecting option K, you see this prompting message:

SAVE-GIVE FILE,START,END[,INIT,RUN]

Your first series of entries should be a filename, a starting address and an ending address—both addresses using a hexadecimal format. If you choose to ignore the portion of the prompting message that is enclosed in brackets, there will be no automatic run when the program is loaded at some later time.

Responding with hexadecimal addresses in the INIT and RUN locations will allow the system to begin execution of the machine-language routines the moment it is loaded under DOS menu item L.

INIT is the starting address of the first of two possible machine-language routines; and if that routine concludes with an RTS, the system will then begin execution at the address specified by RUN.

If INIT and RUN happen to have the same address, only INIT need be specified when the program is saved. In any event, include the commas, but do not include the brackets in your entry.

Loading and Running Binary Programs

Electing DOS menu option L, BINARY LOAD, will bring up this prompting message:

LOAD FROM WHAT FILE?

Respond with a valid filename of a binary program that was saved at some earlier time. If the program was saved in a fashion that forces the system to run it immediately, it will do so. Otherwise, you must elect DOS menu item M.

Selecting item M, RUN AT ADDRESS, you see this prompting message:

RUN FROM WHAT ADDRESS?

Respond by entering the starting address of the binary program. The entry must be in a hexadecimal format.

OPENING AND CLOSING IOCB CHANNELS

The ATARI Operating System has eight separate channels that are organized into I/O control blocks, or IOCBs. One is fully dedicated to the normal screen/keyboard operations, and two others are automatically opened and closed as the operating system dictates. The remaining five IOCBs are free for custom applications. Table 6-1 summarizes the IOCB channels and their allocation.

The operating system automatically closes all channels, except Channel 0, during its normal start-up routine. In order to use one of the channels, then, you must execute a statement that opens it; and there are two BASIC statements that can serve that purpose:

OPEN #*chan*,*task*,*aux1*,"*dev*"
and

XIO 3,#*chan*,*task*,*aux1*,"*dev*"

where *chan* is the IOCB channel to be opened, *task* is a code number representing a task to be performed, *aux1* is a code number for a secondary task description, and *dev* is a device-type specifier.

**Table 6-1. Summary of IOCB Channels
and Their Allocation**

Channel	Allocation
0	Always open for screen editor (E:)
1	Free to use
2	Free to use
3	Free to use
4	Free to use
5	Free to use
6	Automatically opened and closed for graphics operations
7	Automatically opened and closed for I/O operations to a program recorder or disk drive

Chart 6-1 summarizes the device types that can be specified for I/O operations. Notice that the specifiers must end with a colon.

Chart 6-1. Summary of IOCB Device Types, Dev

Device-Name Expressions	
Program recorder C:	
Disk drive D[n]:filename[.ext] Where <i>n</i> is the optional drive number (1-4) <i>filename</i> is a mandatory filename <i>.ext</i> is an optional filename extension Filenames can be composed of up to eight letters and numerals, but must begin with a letter. Extensions can be composed of up to three letters and numerals, but must be preceded with a period. Examples: D:FOAM References filename FOAM on the only disk drive connected to the system. D2:RATS.BAS References filename RATS with extension BAS on disk drive 2. NOTE: DOS must be booted in order to use this device expression.	
Screen Editor E:	
Console Keyboard K:	
ATARI Printer P:	
RS-232 Serial Port R[n]: Where <i>n</i> is the serial-port number (1-4 if used with the 850 serial interface module). Omitting <i>n</i> implies serial port 1. NOTE: The serial device handler, loaded through AUTORUN.SYS, must be resident in memory before this device expression can be used.	
Video Screen S:	

Table 6-2 first lists the general task codes and then cites some examples that apply to particular devices. Obviously, not all possible tasks apply equally well to all types of devices. A write, or output, operation to the program recorder is a meaningful task, but that sort of operation has no meaning at all for keyboard operations.

Table 6-2. Summary of IOCB Task Codes for BASIC's OPEN Statement

General Task Codes

- 1 Append
- 2 Disk operation
- 4 Input
- 8 Output
- 16 Text-window flag
- 32 No screen clear

Examples:

- 12 Input/Output task
- 9 Output with append
- 6 Read disk directory

Useful Program-Recorder Tasks

Task Code

Operation

4

Read from program recorder

8

Write to program recorder

Useful Disk-File Tasks

Task Code

Operation

4

Read from disk

6

Read disk directory

8

Write new file to disk

9

Append disk file

Useful Screen-Editor Tasks

Task Code

Operation

8

Write to screen

9

Append screen

12

Keyboard input/screen output

13

Screen input and output

Useful Keyboard Task

Task Code

Operation

4

Read from keyboard

Table 6-2—cont. Summary of IOCB Task Codes for BASIC's OPEN Statement

Useful Printer Task	
Task Code	Operation
8	Write to printer
Useful RS-232 Port Tasks	
Task Code	Operation
4	Block read
5	Concurrent read
8	Write entire block
9	Concurrent write
13	Concurrent read and write
Useful Screen Tasks	
Task Code	Operation
8	Clear the screen, use no text window, and allow write only
12	Clear the screen, use no text window, and allow both read and write
24	Clear the screen, use a text window, and allow write only
28	Clear the screen, use a text window, and allow both read and write
40	Do not clear the screen (except Mode 0), use no text window, and allow write only
44	Do not clear the screen (except Mode 0), use no text window, and allow both read and write
56	Do not clear the screen (except Mode 0), use a text window, and allow write only
60	Do not clear the screen (except Mode 0), use a text window, and allow both read and write

The *aux1* parameter in the OPEN and XIO statements refer to device-specific operations. There is nothing general about them. Table 6-3 indicates the *aux1* codes as they apply to the various device types. Notice that the value is normally zero, with the notable exception of display-screen applications.

**Table 6-3. Summary of Device-Specific Aux 1
Codes for BASIC's OPEN Statement**

Code		For Program Recorder Operation
0	128	Normal inter-record delay Short inter-record delay
		For Disk Drive
		Always 0
		For Screen Editor
		Always 0
		For Keyboard
		Always 0
Code		For ATARI Printer Operation
0	83	Normal printing Sideways printing (ATARI 820 only)
		RS-232 Ports
		Always 0
Code		For Display Screen Operation
0		BASIC Mode-0 screen
1		BASIC Mode-1 screen
2		BASIC Mode-2 screen
3		BASIC Mode-3 screen
4		BASIC Mode-4 screen
5		BASIC Mode-5 screen
6		BASIC Mode-6 screen
7		BASIC Mode-7 screen
8		BASIC Mode-8 screen

Consider this OPEN statement:

OPEN #2,4,0,K:

That statement literally says: open IOCB Channel 2 for reading operations from the keyboard. An equivalent XIO version of the same thing is:

XIO 3,#2,4,0,K:

After executing either of those statements, you can fetch a character from the keyboard by executing `INPUT #3, numvar` or `GET #3, numvar`; where *numvar* will take on the ASCII code of the current keystroke.

The operating system normally uses IOCB Channel 6 for the screen, but you can open another channel for graphics operations:

`OPEN #4,40,4,S:`

That happens to open IOCB Channel 4 for a version of screen Mode 4 that does not clear and has no text window.

Although the operating system automatically closes all opened channels when it successfully comes to an end, it is a good idea to keep things tidy by closing the channels you use when you are through with them. There are two ways to go about doing that:

`CLOSE #chan`

`XIO 12,#chan,0,0,"dev"`

USING THE XIO COMMAND

The XIO command is one of the most powerful of ATARI BASIC's commands. It can be a rather complicated command, but it can be used for setting up virtually any kind of I/O task through the ATARI system's available I/O control blocks.

The previous section in this chapter demonstrated the fact that properly formatted XIO commands can replace the OPEN and CLOSE commands. A brief look at Table 6-4 shows that it can simulate a lot of other I/O-related BASIC commands.

The XIO command has this general syntax:

`XIO cmd,#chan,aux1,aux2,"dev"`

where *chan* and *dev* are identical to their counterparts in the OPEN statement, and Table 6-4 shows the *cmd* code and their meaning.

Table 6-4. Summary of XIO Commands, CMD

For General Operations		
Code	Operation	BASIC Equivalent
3	Open a channel	OPEN
5	Input a line	INPUT
7	Input a character	GET
9	Output a line	PRINT
11	Output a character	PUT
12	Close a channel	CLOSE
13	Get current IOCB status	STATUS
17	Draw a line (graphics)	DRAWTO
18	Fill an area (graphics)	none
For Disk Operations		
Code	Operation	BASIC Equivalent
32	Rename a file	DOS menu item E
33	Delete a file	DOS menu item D
35	Lock a file	DOS menu item F
36	Unlock a file	DOS menu item G
37	Move the file pointer	POINT
38	Find the file pointer	NOTE
254	Format a disk	DOS menu item I
For RS-232 Serial Operations		
Code	Operation	
32	Output short block	
34	Set outgoing lines DTR, RTS, and XMT	
36	Set baud rate, word size, stop bits, and ready monitoring	
38	Set translation modes and parity	
40	Start concurrent I/O mode	

The *aux1* and *aux2* parameters are set to 0 under all but the following XIO commands: XIO 3, XIO 34, XIO 36 and XIO 38. The XIO 3 command simulates the OPEN statement, using *OPENS task* and *aux1* arguments (Chart 6-1 and Table 6-2) for XIO's *aux1* and *aux2*, respectively. This chapter concludes with descriptions of the special arguments for XIO 34, XIO 36 and XIO 38.

Controlling Outgoing Lines With XIO 34

The ATARI 850's serial output can be configured for Data Terminal Ready (DTR), Request To Send (RTS) and data Transmit (XMT) lines. Serial can handle all three, ports 2 and 3 have the DTR and XMT options, while port 4 has only XMT.

The general form of the XIO 34 command thus looks like this:

XIO 34,#chan,aux1,0,"R[n]:"

where *chan* is the IOCB that is handling the data, and *n* is the serial port number. The *aux1* argument determines the DTR/RTS/XMT configuration as summarized in Table 6-5.

**Table 6-5. XIO 34 Aux 1 Values. Set
Aux 2 to 0 in All Instances**

DTR	RTS	XMT	Code
Off	Off	Off	162
Off	Off	On	163
Off	On	Off	178
Off	On	On	179
On	Off	Off	226
On	Off	On	227
On	On	Off	242
On	On	On	243

Configuring Baud Rate, Word Size, and Stop Bits With XIO 36

The XIO 36 statement must be used for specifying serial-port operations with regard to the number of stop bits, the word size, transmission baud rate, and incoming-signal tests.

The general form of this XIO statement is

XIO 36,#chan,aux1,aux2,"R[n]:"

Table 6-6 shows the values for *aux1* that are to be derived by summing one value from each of the three columns. If, for example, you are configuring the port of 1 stop bit (value = 0), a 7-bit word size (value = 16) and a baud rate of 1200 (value = 10), the final value of (*aux1*) is $0 + 16 + 10 = 26$.

The *aux2* argument sets the tests performed on the incoming signal. The options are Data Set Ready (DTS), Clear to Send (CTS), Carrier Detect (CRX), or any combination of them. Table 6-6 also summarizes the values that correspond to those testing features.

Setting Translation Modes and Parity with XIO 38

The XIO 38 command handles two rather independent features of the RS-232 scheme. One feature exploits the fact that the 850 and serial device handler can send and receive data of opposite or like parity. See the Input and Output Parity columns in Table 6-7.

The second feature expressed by the XIO 38 command has to do with the unusual character coding scheme for the ATARI system. As long as you are communicating between ATARI systems, the built-in ATASCII coding causes no problems; and there is no need for an appended line feed nor any translation. But the differences between standard ASCII and ATASCII can become troublesome when communicating between an ATARI system and another device that uses standard ASCII coding. The purpose of the translation feature is to smooth out the significant differences.

Table 6-6. XIO 36 Values for Aux 1 and Aux 2

aux 1					
Stop Bits	Value	Word Size	Value	Baud Rate	Value
1	0	8	0	300	0
2	128	7	16	45.5	1
		6	32	50	2
		5	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

aux 2			
DSR	CTS	CRX	Value
No	No	No	0
No	No	Yes	1
No	Yes	No	2
No	Yes	Yes	3
Yes	No	No	4
Yes	No	Yes	5
Yes	Yes	No	6
Yes	Yes	Yes	7

Table 6-7. XIO 38 Values for Aux 1. Set Aux 2 to Zero Unless Using the Heavy Translation Mode

Line Feed		Translation		Input Parity		Output Parity	
No	0	Light	0	Ignore	0	Same	0
Yes	64	Heavy	16	Odd	4	Odd	1
		None	32	Even	8	Even	2
				Ignore	12	Bit on	3

The values shown in each of the four columns in Table 6-7 should be selected and summed to arrive at a final value for *aux1* in the XIO 38 command.

Working under light translation, the system changes the EOL character to the standard CR and sets the most-significant bit of each ATASCII code to zero. The result is something that is quite close to standard ASCII. As far as incoming data is concerned, the standard ASCII CR character is translated to the ATARI system's equivalent, EOL, and the most-significant bit of each character code is ignored.

Under the heavy-translation transmission, a light translation is performed, but then only those ATASCII characters that correspond directly to equivalent ASCII codes are sent. And during the input operations, the system changes ASCII's CR character to EOL, passes those characters whose ASCII and ATASCII codes are identical. If the codes aren't identical, the system substitutes the ATASCII character code that is specified by the *aux2* parameter of the XIO 38 command.

Chapter 7

A Miscellany of Principles and Procedures

Whenever authors and editors are organizing a book of this sort, it seems that there are always a few topics that do not fit neatly into the overall presentation. That has been the case with this book, and the purpose of this chapter is to discuss some of those topics.

MORE ABOUT THE SOUND FEATURES

The ATARI system's sound features are about as unique in the world of personal computing as its color graphics features are. For a programmer with the proper interest and motivation, the sound features offer a wealth of fascinating opportunities—from renditions of 4-part musical scores to complex sound effects.

Working under ATARI BASIC, the SOUND statement can select one of four voices, or sound channels, set the pitch of the sound, the amount of distortion and the loudness for the selected voice. It is possible to work with all four voices simultaneously; and all of that sound is reproduced at the loudspeaker in the tv receiver or monitor.

The SOUND Statement

ATARI BASIC's SOUND statement has this general syntax:

SOUND *voice*,*pitch*,*dist*,*volume*

where *voice* is the sound channel (0 through 3) being addressed, *pitch* is the frequency code (0 is highest and 255 is lowest), *dist* is an even-numbered distortion value (0 for maximum noise, or distortion, and 14 for pure tones), and *volume* is the volume level (0 for no sound, and 15 for maximum loudness).

The SOUND statement does not include any provisions for determining the duration of the sound; once a SOUND statement is executed, it produces its specified sound until some other action causes it to change or stop. In fact, it is sometimes more troublesome to stop a sound than it is to initiate it. If a short SOUND program includes no provisions for turning off the sounds and it does not conclude with an END statement, the system will continue generating the prescribed sounds even after the program is done. One way to silence the system in that case is to strike the **SYSTEM RESET** key. Or better, yet, conclude the program with an END statement.

There are some other BASIC statements that automatically silence any sound: CLOAD, CSAVE, DOS, ENTER, LOAD, NEW, RUN and SAVE. The most elegant way to silence a particular voice, however, is by specifying zeros for the tone, distortion and volume parameters in its SOUND statement. The following program plays a particular sound from voice 0 for a short period of time, then silences that voice before coming to an end.

```
10 SOUND 0,193,14,12
20 FOR D=0 TO 100:NEXT D
30 SOUND 0,0,0,0
```

Fig. 7-1 relates some of the *tone* values to an ordinary musical scale. The figure is most helpful when you want to reproduce music that is already printed on a score.

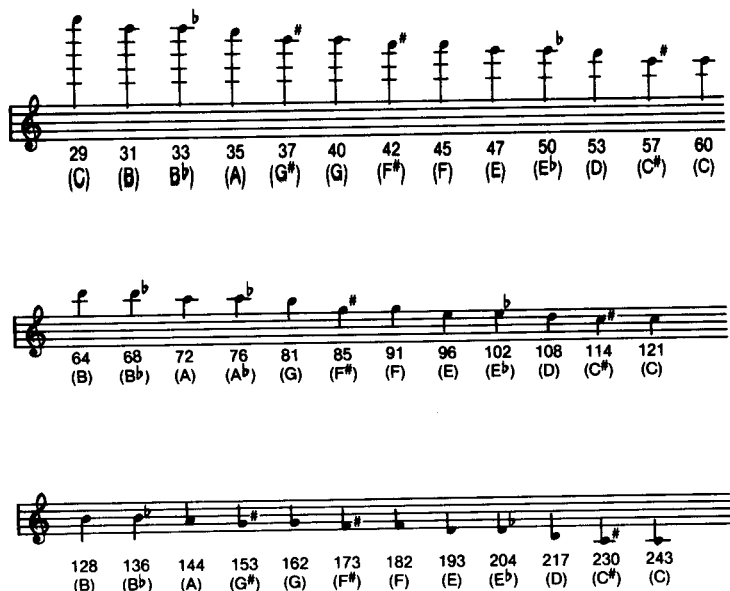


Fig. 7-1. The tone values as they relate to a musical scale.

Here is a routine that sets up some “barbershop” harmony:

```

10 SOUND 0,121,14,12:GOSUB 60
20 SOUND 1,96,14,12:GOSUB 60
30 SOUND 2,81,14,12:GOSUB 60
40 SOUND 3,60,14,12:GOSUB 60
50 GOSUB 60:END
60 FOR D=1 TO 200:NEXT D:RETURN

```

Or if you want a single voice to play those four notes in succession:

```

10 SOUND 0,121,14,12:GOSUB 60
20 SOUND 0,96,14,12:GOSUB 60
30 SOUND 0,81,14,12:GOSUB 60
40 SOUND 0,60,14,12:GOSUB 60
50 GOSUB 60:END
60 FOR D=1 TO 200:NEXT D:RETURN

```


Clearly, the duration of sounds and their timing must be handled by separate timing statements, usually FOR . . . NEXT loops.

Reproducing Musical Scores

Reproducing simple musical scores is a straightforward procedure. As long as all voices are playing notes of the same duration, it is a simple matter of setting up the SOUND statements and executing them for appropriate periods of time.

But matters become more complicated when dealing with scores calling for controlling notes of different duration simultaneously. There are a number of workable ways to approach that situation, but perhaps the most dynamic is to apply a multitasking technique. The general idea is to write a separate musical subroutine for each voice, and then use ON . . . GOSUB statements to cycle through the subroutines until the composition is finished.

Unfortunately, it is beyond the intended scope of this book to pursue that idea any further. It has to be enough to say that anyone with an understanding of program multitasking and the basic principles of music can reproduce compositions of complexity and duration that are limited only by one's patience and creative ability. (RAM capacity is not a serious obstacle if you use some linking techniques to load new segments of programming from a disk as the composition progresses.)

Experimenting With Sound Effects

Whereas it is possible to reproduce musical compositions from a score, matters are not so clearly defined for generating sound effects. Achieving a desired sound is more a matter of trial-and-error experimentation than anything else. A few general principles can guide your work, however.

The distortion parameter can be especially important for sound effects. A bit of experimenting with that parameter, alone, can give you a good feeling for its effects.

Then, too, it is often desirable to create certain sounds by combining two or more voices. Few other brands of personal computers offer that advantage.

The duration of events is, of course, determined by timing loops.

Finally, it can be instructive to work with some attack and decay routines. The general idea is to fit a SOUND statement within a FOR . . . NEXT loop that increments or decrements a variable for the loudness parameter.

MORE ABOUT THE USR FUNCTION

The USR function is ATARI BASIC's route into machine language programs. In its simplest form, the USR function has this syntax:

$$var = \text{USR}(addr)$$

That function initiates the execution of a machine language program that begins at address *addr*. A numerical variable, *var* is meaningless—a “dummy” variable—in this simple context.

The USR function initiates a machine language routine, and an RTS instruction at the conclusion of that routine returns the system to BASIC once again.

Fig. 7-2 shows how the simplest kind of USR function affects the 6502 stack. Prior to executing the USR statement, the stack contains undetermined data that is related to the operation of the system (Fig. 7-2A). Immediately after executing a simple USR function, the computer pushes the BASIC return address and the number of variables in the function to the stack. In this simple case, there are no variables being passed; but nevertheless, that byte of data appears on the top of the stack. See Fig. 7-2B.

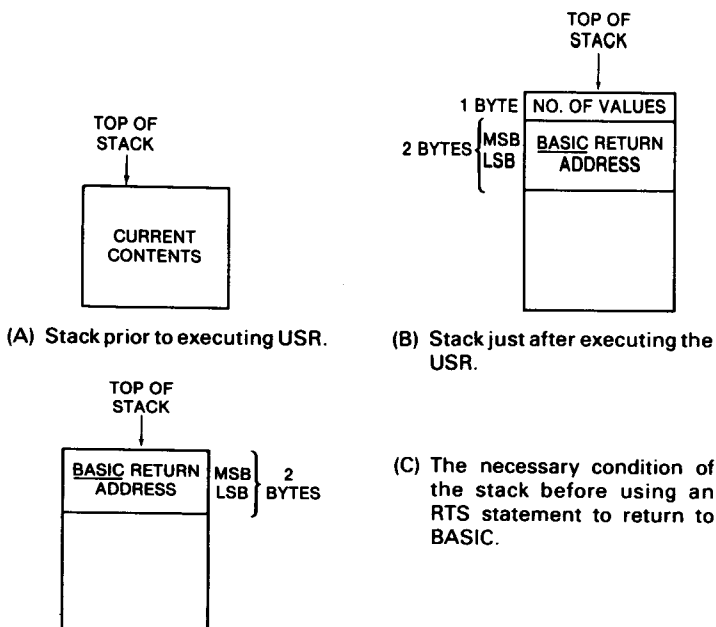


Fig. 7-2. The system stack under the simplest kinds of USR commands.

Before returning to BASIC by executing a machine-language RTS instruction, you must use a PLA instruction to pull that number-of-variables byte off the stack. Fig. 7-2C shows how the stack should appear before executing the RTS instruction.

Thus, any USR statement of this simple form:

`var = USR(addr)`

must call a machine-language routine that ends with these two instructions:

```
PLA    ;Pull no. of variables off the stack
RTS    ;Return to BASIC
```

The following BASIC program first POKES the two-instruction machine code into addresses 1536 and 1537, and then it uses a USR function to execute it.

```

10 POKE 1536,104:POKE 1537 96
20 A=USR(1536)

```

An assembly language version of that two-instruction routine looks like this:

```

1536 104 PLA ;Pull no. of variables from stack
1537 96 RTS ;Return to BASIC

```

It is a do-nothing routine, but it does illustrate the need for pulling the number-of-values byte off the stack before executing the RTS instruction.

The next program uses a machine language routine to print the ATARI character set, including the inverse versions, along the top of the screen:

```

10 FOR N=1536 TO 1549
20 READ X:POKE N,X
30 NEXT N
40 GRAPHICS 0
50 POSITION 0,15
60 A=USR(1536)
70 DATA 162,0,138,157,64,156
80 DATA 232,138,201,128,208,246
90 DATA 104,96

```

An assembly language rendition of the routine that is POKEed into addresses 1536 through 1549 looks like this:

```

1536 162 0 LDX#0 ;Zero the X register
1538 138 TXA ;Transfer X to A
1539 157 64 156 STA 40000,X;Plot A to the
screen
1542 232 INX ;Increment X
1543 138 TXA ;Transfer X to A
1544 201 128 CMP#128 ;Counting done?
1546 208 246 BNE - 10 ;If not, plot again
1548 104 PLA ;Get rid of no. var.
1549 96 RTS ;Return to BASIC

```

Passing Values to a Machine Routine

The USR function includes provisions for passing any number of numerical values to a machine language routine. The general syntax in this case is:

var = USR(*addr*,*expr*)

where *addr* is the starting address of the machine language programming, *expr* is a numeric value or expression to be passed to the machine language routine, and *var* is (in this context) a “dummy” variable. The value of *expr* must be a positive integer between 0 and 65536.

The value is passed to the machine language routine through the 6502 stack, and Fig. 7-3 shows how the stack appears before the USR function is executed, immediately after the function is executed, and how it should appear just prior to executing the RTS instruction to return to BASIC.

Notice from the diagram that the number-of-values byte appears on the top of the stack. After that comes the MSB of the value *expr*, followed by the LSB of *expr*. The return address of the BASIC program is the final item in the stack.

NOTE: The MSB of a value passed to a machine language routine appears on the stack before its LSB does.

The value that is passed to the stack is thus buried under the number-of-values byte; so in order to do anything useful with the value, the machine language routine must first execute a PLA to get rid of that byte. Then the value—MSB followed by LSB—is available via two more PLA instructions.

The following BASIC program first POKes a machine language routine into addresses 1536 through 1543. It then prompts the user to enter a positive integer between 0 and 65536, executes an INPUT statement to assign the value to variable N, and then executes a USR function that both initiates the routine and passes the value of N to it.

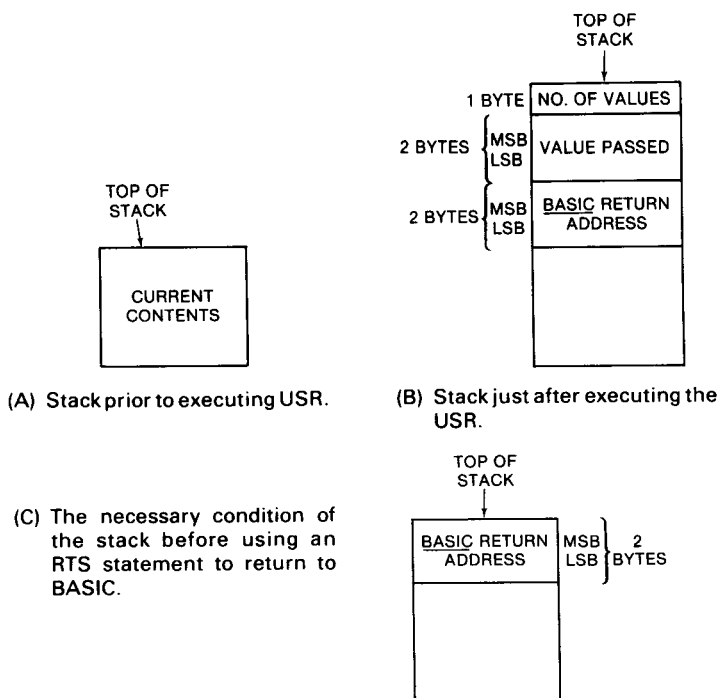


Fig. 7-3. The system stack when passing a single variable under USR commands.

The program returns to BASIC at line 90 where it prints a pair of values from zero-page memory addresses 203 and 204. And what you see there is a 2-byte decimal version of the value assigned to N.

```

10 FOR N=1536 TO 1543
20 READ D:POKE N,D
30 NEXT N
50 GRAPHICS 0
60 PRINT "ENTER A POSITIVE INTEGER VALUE
  (0-65536)"
70 INPUT N
80 X=USR(1536,N)
90 PRINT PEEK(203),PEEK(204)
100 DATA 104,104,133,204,104,133,203,96

```

The assembly version of the machine language routine looks like this:

1536	104	PLA	;Throw away no. of vals. byte
1537	104	PLA	;Fetch MSB of N from stack
1538	133	STA 204	;Store it in 204
1540	104	PLA	;Fetch LSB of N from stack
1541	133	STA 203	;Store it in 203
1543	96	RTS	;Return to BASIC

Whenever it is necessary to pass more than one value to a machine language program, the USR statement looks like this:

var = USR(*addr*,*expr1*,*expr2*, . . .)

The multiple expressions are passed to the stack as 2-byte integer values, and their MSB appears on the stack before their LSB does. What's more, the values are placed on the stack in reverse order: *expr1* will be buried deeper in the stack than any other expressions that follow it in the USR function. Fig. 7-4 illustrates how the stack should be handled whenever the USR function passes more than one value to it.

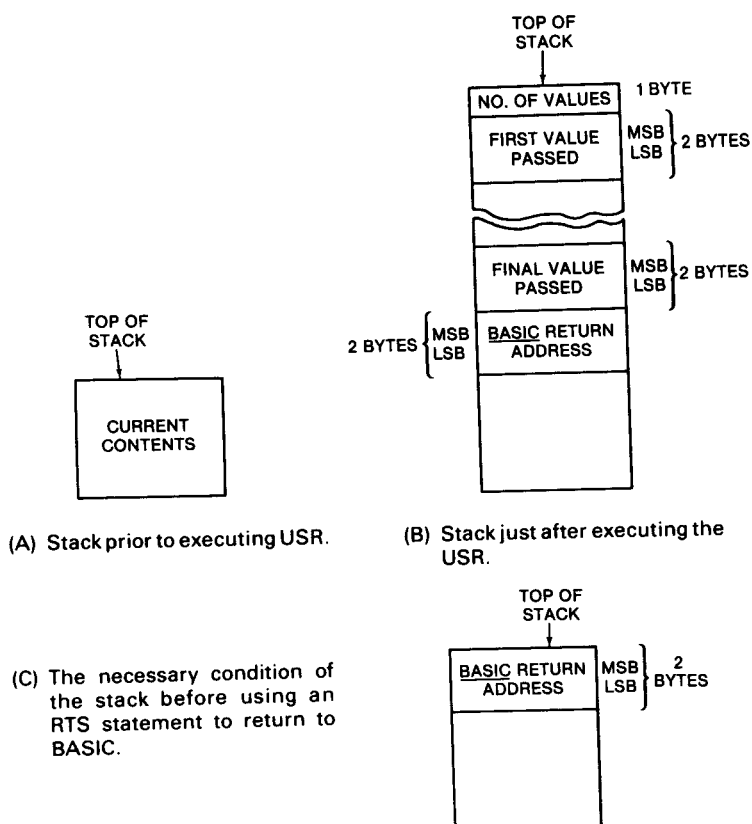


Fig. 7-4. The system stack when passing multiple variables under USR commands.

The following version of the USR function passes the length and address location of a string, M\$:

$X = \text{USR}(1536, \text{LEN}(\text{M\$}), \text{ADR}(\text{M\$}))$

Having done that, the number-of-values byte will appear on the top of the stack, followed by the results of the LEN function, the results of the ADR function, and the returning address for BASIC. As in the earlier instances, the MSB for a given value appears on top of its LSB.

Passing Values From a Machine Language Routine

The previous discussions treat the USR function in such a way that the var is a simple dummy variable. It takes on some meaning, however, when a machine language routine is supposed to pass a numerical value back to basic. Consider this portion of a BASIC program that loads and executes a machine language routine:

```
100 X=USR(1536)
110 PRINT X
```

The idea is to initiate some sort of machine language routine from address 1536. And assuming that the routine generates a meaningful numerical value, it can be returned to BASIC as assigned to variable X. In order to do that, however, the machine language routine must save the value in address locations 212 and 213 (LSB followed by MSB).

Alternatively, you can return values from machine language to BASIC by storing the values to some well-defined RAM locations; and after returning to BASIC, fetch the values by means of PEEK statements.

SCREEN DISPLAY LISTS

The ATARI system's ANTIC hardware device is actually a small microprocessor in its own right; and like a microprocessor, it can be programmed to perform some special tasks.

The primary purpose of ANTIC is to control all the features of the display, and those features are developed by the special ANTIC programming, called the *display list*.

The ANTIC Instruction Set

The ANTIC instruction set includes codes for leaving a designated number of blank lines on the screen, setting up 14 different kinds of screen formats, and two kinds of address jumps.

Chapter 4 describes eight basic screen formats that are directly accessible from BASIC. ANTIC actually allows 14 screens; but of equal importance is the fact that ANTIC uses different screen-number assignments than BASIC does. Table 7-1 lists all available screen formats.

Table 7-1. ANTIC'S Screen Modes

ANTIC Mode	BASIC Mode	Pixels (horiz.)	Bytes per Line	TV Lines per Pixel	Bits per Pixel
2	0	40	40	8	8
3	none	40	40	10	8 Character Modes
4	none	40	40	8	8
5	none	40	40	16	8
6	1	20	20	8	8
7	2	20	20	16	8
8	3	40	10	8	2
9	4	80	10	4	1 Bit-Mapped
10	5	80	20	4	2 Graphics Modes
11	6	160	20	2	1
12	none	160	20	1	1
13	7	160	40	1	2
14	none	160	40	8	2
15	8	320	40	16	1

Notice that BASIC's Mode-0 screen is called mode 2 as far as ANTIC is concerned. Also notice that ANTIC offers five additional screen modes. The columns of data associated with each mode indicate the number of pixels per line (under normal screen width), the number of screen data bytes per line, the number of tv vertical scan lines per pixel, and the number of screen data bits per pixel. As described later, the number of tv vertical scan lines per pixel is an especially important number when composing the display list.

Chart 7-1 shows the basic display list instruction set. The instruction set actually uses most of the decimal values between 0 and 255, but the ones shown here represent the starting points.

Chart 7-1. ANTIC Display List Instruction Set

HSCROL	N	Y	N	Y	N	Y	N	Y	
VSCROL	N	N	Y	Y	N	N	Y	Y	
LMS	N	N	N	N	Y	Y	Y	Y	
BLK 1	0	0	0	0	0	0	0	0	Blank lines
BLK 2	16	16	16	16	16	16	16	16	
BLK 3	32	32	32	32	32	32	32	32	
BLK 4	48	48	48	48	48	48	48	48	
BLK 5	64	64	64	64	64	64	64	64	
BLK 6	96	96	96	96	96	96	96	96	
BLK 7	112	112	112	112	112	112	112	112	
BLK 8	128	128	128	128	128	128	128	128	
JMP	1	1	1	1	1	1	1	1	Jumps
JVB	65	65	65	65	65	65	65	65	
CHR 2	2	18	34	50	66	82	98	114	BASIC Mode 0
CHR 3	3	19	35	51	67	83	99	115	
CHR 4	4	20	36	52	68	84	100	116	
CHR 5	5	21	37	53	69	85	101	117	
CHR 6	6	22	38	54	70	86	102	118	BASIC Mode 1
CHR 7	7	23	39	55	71	87	103	119	
MAP 8	8	24	40	56	72	88	104	120	BASIC Mode 3
MAP 9	9	25	41	57	73	89	105	121	BASIC Mode 4
MAP 10	10	26	42	58	74	90	106	122	BASIC Mode 5
MAP 11	11	27	43	59	75	91	107	123	BASIC Mode 6
MAP 12	12	28	44	60	76	92	108	124	BASIC Mode 7
MAP 13	13	29	45	61	77	93	109	125	
MAP 14	14	30	46	62	78	94	110	126	
MAP 15	15	31	47	63	79	95	111	127	

NOTE: Add 128 to ANTIC Opcodes to obtain the corresponding Instruction Interrupt codes.

The BLNK instructions indicate the number of successive blank lines that are to appear on the screen, and the ANTIC instructions dictate the screen mode to be applied for the current number of vertical scan lines. Referring to Table 7-1, for example, ANTIC mode 10 uses 4 scan lines per pixel; so an ANTIC 10 display-list instruction will apply that mode to 4 successive tv scan lines. And if you happen to want 8 such lines in succession, the display list must show two ANTIC 10 instructions in succession.

The only difference between the two jump instructions is that JVB waits for the next tv vertical retrace before it is executed. JMP does not. Both instructions must be followed by a 2-byte destination address.

JVB is always used at the end of a display list; it usually jumps the ANTIC programming back to the beginning of the display list. It is altogether possible to interweave different kinds of screen formats, however, by doing a JVB to the beginning of a different display-list program.

The simpler JMP instruction is used only when the display-list addressing crosses a 1k boundary—something most programmers prefer to avoid in the first place.

Structure of a Display List

As in the case of any sort of microprocessor programming, display-list programming for the ANTIC chip must follow some strict rules.

First, every display list must begin by specifying 24 consecutive blank lines. This requirement eliminates difficulties that are associated with tv vertical overscan. Using the fewest possible number of instructions, the 24 lines can be specified by three BLNK 8 instructions in succession:

112
112
112

The second step in the program is to point to the starting address of the screen RAM to be used for the project. That is a matter of loading ANTIC's memory-screen counter with that address; and that is done by invoking the LMS option.

The LMS option is invoked by summing a value of 64 with the ANTIC mode instruction to be used next. So if you are setting up ANTIC mode 10, invoke the LMS by means of instruction code $10 + 64$, or 74.

But that isn't all. The LMS instruction must be followed by a 2-byte version of the starting address of the screen RAM. If you are planning to set the screen RAM from address 40960, the LMS instruction must be followed by 0 (the LSB) and 160 (the MSB).

Thus far, the display list programming looks like this:

```
112    BLNK 8
112    BLNK 8
112    BLNK 8
  74    LMS, ANTIC 10
   0    (to address 40960)
160
```

It specifies the initial 24 blank lines, establishes the beginning of the screen RAM at address 40960, and draws the first 4 vertical scan lines of an ANTIC mode 10 screen.

In the simplest case, the third step is to continue specifying ANTIC mode instructions until the *total* number of vertical scan lines (including blanks) is equal to 192. The example just cited is using ANTIC mode 10—4 scan lines per operation, so adding 41 consecutive ANTIC 10 instructions completes the screen.

NOTE: A display list must account for no more than 192 vertical scan lines—the number of lines required for filling the screen. It is possible to use fewer lines to shorten the screen, but using more than 192 lines causes undesirable video effects.

After accounting for all of the lines, usually 192 of them, the final step is to apply the JVB instruction. The purpose is to wait for the next vertical blanking interval, and then loop the display list back to the beginning. That is a 3-byte instruction that begins with the JVB op code, 65, followed by a 2-byte rendition of the RAM starting address of the display list.

So if you decide to locate your display list at address 32768 (it is generally located just below the screen RAM area), then the final three instructions look like this:

```
65    JVB
   0    (to 32768)
144
```


The beginning of the ANTIC Mode 7 text window is marked by an LMS instruction that points to the beginning of the screen addressing for that part of the display—to address 40800. The program concludes with three additional ANTIC 7 lines and a JVB back to the beginning of the display list at 40526.

That is a good example of a mixed-mode display list.

There are two additional rules that must apply in all instances:

1. A JMP instruction must be used wherever the display-list programming must cross a 1k boundary value.
2. An LMS instruction must be used wherever the screen addressing crosses a 4k boundary address.

Both situations can be avoided by being very selective about the choice of starting addresses for the display list and screen RAM. They most often arise, however, when it is necessary to conserve RAM and, of course, when using very long and complex display lists or large screen RAM areas.

Other display list options include horizontal scrolling (HSCROL), vertical scrolling (VSCROL) and display interrupts (INT). They are invoked in a fashion similar to LMS—by summing certain values with the current ANTIC mode or BLNK code (see Chart 7-2).

Locating the Display List

The display-list programming generally takes up a lot less RAM space than the screen RAM will. Programmers thus tend to specify the screen RAM area first, avoiding a 4k boundary value wherever possible.

The display list, more by convenience and convention than necessity, is generally located below and as close as possible to the beginning of the screen RAM. And, where feasible, the display-list addressing should not cross a 1k boundary.

Chart 7-2. Summary of Display-List Options

For mode instructions—mode=ANTIC+HSCROL+VSCROL+LMS+INT

For blank instructions—blank=BLNK+INT

Where:

mode=op code for a mode line

blank=op code for a blank line

ANTIC=the current ANTIC mode number (Table 7-1)

BLNK=the current BLNK number (Table 7-1)

HSCROL: 0=no horizontal scrolling
16=horizontal scrolling

VSCROL: 0=no vertical scrolling
32=vertical scrolling

LMS: 0=LMS not invoked
64=LMS invoked

INT: 0=no interrupt
128=interrupt invoked

The starting address of the screen RAM then becomes part of the display list—inserted by means of an LMS instruction. The starting address of the display list, itself, usually appears as part of the JVB instruction at the end of the display list.

Once the display list is loaded into RAM, it is necessary to POKE its starting address into SDSTL, addresses 560 and 561; but that can be done properly only if the DMA control (address 559) is turned off during the POKE intervals. The routine generally takes this form:

POKE 559,0 (disable DMA)

POKE 560,lsb of list address

POKE 561,msb of list address

POKE 559,34 (enable DMA for standard playfield)

A LOOK AT TOKENIZED BASIC

BASIC is an interpretive language. That is, the computer reads each instruction in the program just prior to using it. It follows, then, that programs using shorter statements can be executed faster and loaded into smaller amounts of

RAM. A system that would have to read every character in every line of a program would be terribly slow-acting and take up a great deal of RAM space.

BASIC gets around that difficulty by translating every valid command, statement, function and operator into a 1-byte token. Variable names and constant values are still carried in the usual ATASCII forms, but virtually everything else is tokenized. See the list of ATARI BASIC tokens in Tables 7-3 through 7-5.

Table 7-3. Command Tokens

Code	Command	Code	Command
0	REM	28	POINT
1	DATA	29	X10
2	INPUT	30	ON
3	COLOR	31	POKE
4	LIST	32	PRINT
5	ENTER	33	RAD
6	LET	34	READ
7	IF	35	RESTORE
8	FOR	36	RETURN
9	NEXT	37	RUN
10	GOTO	38	STOP
11	GO TO	39	POP
12	GOSUB	40	?
13	TRAP	41	GET
14	BYE	42	PUT
15	CONT	43	GRAPHICS
16	COM	44	PLOT
17	CLOSE	45	POSITION
18	CLR	46	DOS
19	DEG	47	DRAWTO
20	DIM	48	SETCOLOR
21	END	49	LOCATE
22	NEW	50	SOUND
23	OPEN	51	LPRINT
24	LOAD	52	CSAVE
25	SAVE	53	CLOAD
26	STATUS	54	implied LET
27	NOTE	55	ERROR - (syntax)

Table 7-4. Operator Tokens

Code	Operator	Code	Operator
14	num constant	37	+
15	string constant	38	-
16	not used	39	/
17	not used	40	NOT
18	,	41	OR
19	\$	42	AND
20	: (end of statement)	43	(
21	;	44)
22	end of line	45	= (num assign)
23	GOTO	46	= (str assign)
24	GOSUB	47	<= (string)
25	TO	48	<>
26	STEP	49	>=
27	THEN	50	<
28	#	51	>
29	<= (numeric)	52	=
30	<>	53	+ (unary)
31	>=	54	-
32	<	55	((as in a string)
33	>	56	((array)
34	=	57	((array)
35		58	((function)
36	*	59	((dimension)
		60	, (as in an array)

Unless directed otherwise, BASIC programs are saved on tape or disk in this tokenized form. However, Chapter 6 describes conditions where it is more desirable to deal with a nontokenized version—a version where every character in every BASIC statement is literally saved as an ATASCII character.

Table 7-5. Function Tokens

Code	Function
61	STR\$
62	CHR\$
63	USR
64	ASC
65	VAL
66	LEN
67	ADR
68	ATN
69	COS
70	PEEK
71	SIN
72	RND
73	FRE
74	EXP
75	LOG
76	CLOG
77	SQR
78	SGN
79	ABS
80	INT
81	PADDL
82	STICK
83	PTRIG
84	STRIG

Chapter 8

The ATARI Memory Map

Fig. 8-1 shows the general memory map for the ATARI system. It is divided into two main parts: a RAM area between addresses 0 and 49152, and a ROM area from 49152 through 65535.

The ROM area is common to all systems, regardless of the amount of RAM that is installed. The minimum RAM area (8k) is situated between addresses 0 and 8192. The material in this chapter deals most extensively with those two portions of the ATARI memory map.

Thus, any working ATARI system has at least 8k of RAM installed at the lower end of the memory map. A 16k system includes that RAM area plus an additional 8k of RAM between addresses 8192 and 16383. A 32k system would have RAM available up to address 32767.

A 48k system has RAM available through address 49151. Notice from the overall memory map that the upper 16k of RAM in such a system overlaps the left- and right-cartridge areas. The cartridges are actually ROM devices; and in a 48k system, the cartridge ROM operations take full precedence over any RAM that they overlay.

So if you happen to have a 48k system and install a cartridge in the left-cartridge area, you will still have some RAM available in the right-cartridge area (between addresses 32768 and 40959).

Table 8-1 summarizes the RAM area that is generally available to the user when the BASIC cartridge is installed.

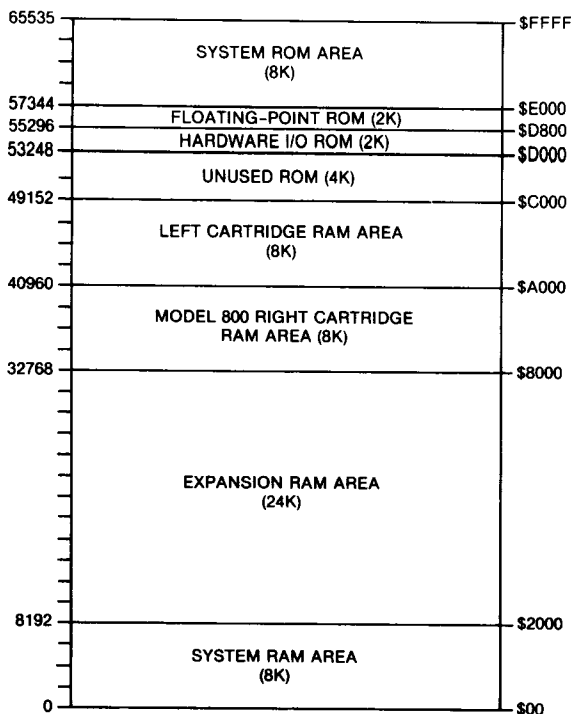


Fig. 8-1. Overall memory map of the ATARI Home Computer system.

Table 8-1. Range of RAM Addresses Available to the User as a Function of the Amount of Installed RAM. It Is Assumed That the Standard ATARI Home Computer BASIC Cartridge is Installed

RAM Size	RAM Address Range Available	
	Decimal	Hexadecimal
8k	1536-8191	\$0600-\$1FFF
16k	1536-16383	\$0600-\$3FFF
32k	1536-32767	\$0600-\$7FFF
48k	1536-40959	\$0600-\$9FFF

NOTE: If DOS is installed, the lowest-usable RAM address is raised to 10879 (\$2A7F). DOS thus requires at least a 16k system.

ZERO-PAGE AND STACK RAM

The ATARI system uses a 6502 microprocessor as the central processor unit (CPU); and it is the nature of that device that dictates the application of the RAM area from address 0 through 255 (\$00-\$FF). Using conventional 6502 terminology, the first half of that area is called the zero-page, and it has special significance in terms of machine-language addressing. (See the 6502 instruction set in Chapter 9.)

The second half of this RAM area is normally dedicated entirely to 6502 stack operations; and a programmer should use it only when fully aware of those internal operations.

ATARI Home Computer engineers have divided the zero-page RAM, itself, into two equal sections. The lower section, 0-127, is used for the general operating-system and the remainder, addresses 128-255, is used for BASIC's zero-page operations.

Chart 8-1 outlines the usage of the operating-system portion of the zero-page area, and Chart 8-2 outlines the portion used by BASIC.

**Chart 8-1. Operating System Usage
of the Zero-Page RAM: 0-127**

\$00-\$01	0-1	LNZBS
Application is unclear.		
\$02-\$03	2-3	CASINI
Successful cassette boot initialization data and addresses.		
\$04-\$05	4-5	RAMLO
RAM test pointer during initialization.		
\$06	6	TRAMSZ
Page number of highest available RAM address. Value is passed to RAMTOP at \$6A\$ (106) after initialization is done.		
\$07	7	TSTDAT
RAM-test register; set to 1 when a cartridge is inserted in the right-hand (B) slot.		

**Chart 8-1—cont. Operating System Usage
of the Zero-Page RAM: 0-127**

\$08	8	WARMST
Warm-start flag; it is set to 0 during cold start.		
\$09	9	BOOT?
Successful disk-boot flag.		
\$0A-\$0B	10-11	DOSVEC
Diskette vector; points to start of BASIC programs.		
\$0C-\$0D	12-13	DOSINI
Disk-boot initialization vector.		
\$0E-\$0F	14-15	APPMHI
Lowest RAM address available for screen memory.		
\$10	16	POKMSK
POKEY interrupt mask.		
\$11	17	BRKKEY
BREAK-key flag: 0 = break \$80 (120) = no break		
\$12-\$14	18-20	RTCLOK
Real-time clock.		
\$15-\$16	21-22	BUFADR
Temporary zero-page address pointer for disk buffer.		
\$17	23	ICCOMT
Command table index pointer.		
\$1A-\$1B	26-27	DSKUTL
Disk utility address.		
\$1C	28	PTIMOT
Printer time-out value.		
\$1D	29	PBPNT
Printer buffer index pointer.		
\$1E	30	PBUFSZ
Printer-buffer record size.		
\$1F	31	PTEMP
Temporary register for next character to printer.		
\$20	32	ICHIDZ
Current device handler index number.		

**Chart 8-1—cont. Operating System Usage
of the Zero-Page RAM: 0-127**

\$21	33	ICDNOZ
Current device drive number.		
\$22	34	ICCOMZ
Current device command number.		
\$23	35	ICSTAZ
Last-returned status code.		
\$24-\$25	36-37	ICBALZ ICBAHZ
Command buffer address pointer.		
\$26-\$27	38-39	ICPTLZ ICPTHZ
PUT-byte address pointer.		
\$28-\$29	40-41	ICBL LZ ICBLHZ
Buffer length for GET and PUT operations.		
\$2A-\$2B	42-43	ICAX1Z ICAX2Z
2-byte auxiliary registers; used mainly for I/O control block operations.		
\$2C-\$2F	44-47	ICSPRZ
Temporary-storage registers.		
\$30	48	STATUS
Internal status register.		
\$31	49	CHKSUM
Checksum register for detecting frame-transfer errors.		
\$32-\$33	50-51	BUFRLO BUFRH
Next-byte address pointer for buffer operations.		
\$34-\$35	52-53	BFENLO BFENHI
End-of-buffer byte pointer.		
\$36	54	CRETRY
Command retry counter.		
\$37	55	DRETRY
Device retry counter.		

**Chart 8-1—cont. Operating System Usage
of the Zero-Page RAM: 0-127**

\$38	56	BUFRL
Buffer full flag.		
\$39	57	RECVDN
Receive-done flag.		
\$3A	58	XMTDON
Transmit-done flag.		
\$3B	59	CHKSNT
Checksum byte-sent flag: 0 = not sent \$FF (255) = sent		
\$3C	60	NOCKSM
No-checksum-coming flag: 0 = checksum is coming NOT 0 = no checksum		
\$3D	61	BPTR
Cassette buffer index pointer.		
\$3E	62	FTYPE
Type flag for inter-record gap.		
\$3F	63	FEOF
Cassette end-of-file flag: 0 = not EOF \$FE (255) = EOF		
\$40	64	FREQ
Cassette read/write BEEP count: 1 = play 2 = record		
\$41	65	SOUNDR
Audio sound flag: 0 = no noise		
\$42	66	CRITIC
Critical-code flag.		
\$43-\$44	67-68	ZBUFP
Buffer address pointer.		
\$45-\$46	69-70	ZDRVA
Drive address pointer.		

**Chart 8-1—cont. Operating System Usage
of the Zero-Page RAM: 0-127**

\$47-\$48	71-72	ZSBA
Sector buffer address pointer.		
\$49	73	ERRNO
Error number for disk operations.		
\$4A	74	CKEY
Cassette-boot request flag.		
\$4B	75	CASSBT
Boot flag: 0 = disk boot 1 = cassette boot		
\$4C	76	DSTAT
Display status.		
\$4D	77	ATTRACT
ATTRACT mode timer register/flag. Increments from 0 to \$7F (127) at 4 second intervals after 9 minutes of no key activity. Set to \$FE (254) at the end of that timing interval.		
\$4E	78	DRKMSK
Dark-color mask used during ATTRACT mode to reduce screen luminance. Has a value of \$FE (254) when ATTRACT is inactive, and \$F6 (246) when active.		
\$4F	79	COLRSH
Color-shift mask register used in ATTRACT mode. Has value of 0 when ATTRACT is inactive.		
\$50	80	TMPCHR
Temporary register screen data during transfer.		
\$51	81	HOLD1
Temporary register for number of entries in a display list.		
\$52	82	LMARGN
Screen address of the left text margin. Range: \$0-\$27 (0-39)		
\$53	83	RMARGN
Screen address of the right text margin. Range: \$0-\$27 (0-39)		

**Chart 8-1—cont. Operating System Usage
of the Zero-Page RAM: 0-127**

\$54	84	ROWCRS
Row address of the current cursor position. Range: \$00-\$BF (0-191)		
\$55-\$56	85-86	COLCRS
Column address of current cursor position. Range: \$00-\$13F (0-319)		
\$57	87	DINDEX
Index value of current screen mode.		
\$58-\$59	88-89	SAVMCS
Address of upper-left corner of display.		
\$5A	90	OLDROW
Row address of the previous graphics cursor position.		
\$5B-\$5C	91-92	OLDCOL
Column address of the previous graphics cursor position.		
\$5D	93	OLDCHR
Holds screen character while it is "hidden" by the text cursor.		
\$5E-\$5F	94-95	OLDADR
Screen memory address of current text cursor location.		
\$60	96	NEWROW
Row destination for next LINE/DRAW step.		
\$61-\$62	97-98	NEWCOL
Column destination for next LINE/DRAW step.		
\$63	99	LOGCOL
Logical-line column number for the current cursor position.		
\$64-\$65	100-101	ADRESS
General-purpose 2-byte register.		
\$66-\$67	102-103	MLTTMP OPNTMP
Multipurpose 2-byte register; used with OPEN command and as address pointer.		
\$68-\$69	104-105	SAVADR FRNADR
General-purpose 2-byte register.		

**Chart 8-1—cont. Operating System Usage
of the Zero-Page RAM: 0-127**

\$6A	106	RAMTOP
Page number of highest available RAM address. Used by display handler.		
\$6B	107	BUFCNT
Current logical line-size counter.		
\$6C-\$6D	108-109	BUFSTR
Temporary 2-byte register used with display routines.		
\$6E	110	BITMSK
Temporary register for bit masks.		
\$6F	111	SHFAMT
Amount of pixel shift required to right-justify the data.		
\$70-\$71	112-113	ROWAC
Accumulator for row-counting during plotting operations.		
\$72-\$73	114-115	COLAC
Accumulator for column-counting during plotting operations.		
\$74-\$75	116-117	ENDPT
End point designation for a line being plotted to the screen.		
\$76	118	DELTAR
Change in screen row position (vertical component) of a slope drawing.		
\$77-\$78	119-120	DELTAC
Change in screen column position (horizontal component) of a slope drawing.		
\$79	121	ROWINC
Drawing row direction: \$01 (1) = down \$FF (255) = up		
\$7A	122	COLINC
Drawing column direction: \$01 (1) = right \$FF (255) = left		
\$7B	123	SWPFLG
Split-screen cursor flag: 0 = normal \$FF (255) = swapped		

**Chart 8-1—cont. Operating System Usage
of the Zero-Page RAM: 0-127**

\$7C	124	HOLDCH
Current character code for subsequent SHIFT or CONTROL logic.		
\$7D	125	INSDAT
Cursor-related temporary storage.		
\$7E-\$7F	126-127	COUNTR
Number of steps involved in drawing a line.		

**Chart 8-2.BASIC Usage
of the Zero-Page RAM: 128-255**

\$80-\$81	128-129	LOMEM
Starting address of resident BASIC programming features.		
\$82-\$83	130-131	VNTP
Starting address of BASIC variable table.		
\$84-\$85	132-133	VNTD
Ending address of BASIC variable table.		
\$86-\$87	134-135	VVTP
Variable table pointer.		
\$88-\$89	136-137	STMTAB
Starting address of BASIC statement table.		
\$8A-\$8B	138-139	STMCUR
BASIC statement pointer.		
\$8C-\$8D	140-141	STARP
Address of the end of the string and array table.		
\$8E-\$8F	142-143	RUNSTK
BASIC stack for GOSUB, FOR-NEXT, etc.		
\$90-\$91	144-145	MEMTOP
Top of BASIC programming area.		
\$92-\$D3	146-211	
Reserved for general BASIC operations.		

**Chart 8-2—cont. BASIC Usage
of the Zero-Page RAM: 128-255**

\$D4-\$D9	212-217	FR0
Floating-point register 0.		
\$DA-\$DF	218-223	FRE
Spare floating-point registers.		
\$E0-\$E5	224-229	FR1
Floating-point register 1.		
\$E6-\$EB	230-235	FR2
Floating-point register 2.		
\$EC	236	FRX
Not used.		
\$ED	237	EEXP
Exponent value.		
\$EE	238	NSIGN
Sign of floating-print value.		
\$EF	239	ESIGN
Sign of floating-point exponent.		
\$F0	240	FCHRFLG
First-character flag for floating-point operation.		
\$F1	241	DIGRT
Number of digits to right of decimal point.		
\$F2	242	CIX
Character index for input buffer.		
\$F3-\$F4	243-244	INBUFF
Input buffer pointer for ATASCII text.		
\$F5-\$F6	245-246	ZTEMP1
Temporary registers.		
\$F7-\$F8	247-248	ZTEMP4
Temporary registers.		

**Chart 8-2—cont. BASIC Usage
of the Zero-Page RAM: 128-255**

\$F9-\$FA Temporary registers.	249-250	ZTEMP3
\$FB Angle flag: 0 = radians 6 = degrees	251	RADFLG DEGFLG
\$FC-\$FD Floating-point number pointer.	252-253	FLPTR
\$FE-\$FF Floating-point second number pointer.	254-255	FPTR2

OPERATING SYSTEM AND BASIC RAM

The ATARI Home Computer's main operating system uses a significant portion of the system RAM area; and when the BASIC cartridge is installed, it, too, uses some system RAM. Exploring both of these areas can turn up a wealth of potentially useful information.

Operating System RAM: 512-1151

Chart 8-3 outlines the operating system RAM area. Most of the comments accompanying the addresses and labels are sufficient to define their function for anyone who is familiar with the fundamental internal workings of a computer and is willing to experiment with some PEEKs and POKEs.

Chart 8-3. Operating-System RAM: 512-1151

\$0200-\$0201	512-513	VDSLST
Display-list interrupt vector.		
\$0202-\$0203	514-515	VPRCED
I/O proceed vector.		
\$0204-\$0205	516-517	VINTER
I/O interrupt vector.		
\$0206-\$0207	518-519	VBREAK
BREAK vector.		
\$0208-\$0209	520-521	VKEYBD
Keyboard interrupt vector.		
\$020A-\$020B	522-523	VSERIN
Serial input ready vector.		
\$020C-\$020D	524-525	VSEROR
Serial output ready vector.		
\$020E-\$020F	526-527	VSEROC
Serial transfer complete vector.		
\$0210-\$0215	528-533	VITMR1 VITMR2 VITMR4
POKEY timer vectors.		
\$0216-\$0217	534-535	VIMIRQ
Interrupt-request vector.		
\$0218-\$0219	536-537	CDTMV1
System timer-1 value; decrements with each VBLANK.		
\$021A-\$021B	538-539	CDTMV2
System timer-2 value.		
\$021C-\$0221	540-545	CDTMV3 CDTMV4 CDTMV5
System timers 3, 4, and 5.		
\$0222-\$0223	546-547	VVBLKI
VBLANK interrupt vector.		
\$0224-\$0225	548-549	VVBLKD
Return from VBLANK interrupt vector.		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$0226-\$0227	550-551	CDTMA1
System timer-1 jump address.		
\$0228-\$0229	552-553	CDTMA2
System timer-2 jump address.		
\$022A	554	CDTMF3
System timer-3 time-out flag.		
\$022B	555	SRTIMR
Key-repeat timer.		
\$022C	556	CDTMF4
System timer-4 time-out flag.		
\$022D	557	INTEMP
Temporary integer register.		
\$022E	558	CDTMF5
System timer-5 time-out flag.		
\$022F	559	SDMCTL
DMA enable/control.		
\$0230-\$0231	560-561	SDLSTL SDLSTH
Starting address of display list.		
\$0232	562	SSKCTL
Copy of serial port control register \$D20F (53775).		
\$0233	563	
Not used.		
\$0234	564	LPENH
Light pen horizontal position.		
\$0235	565	LPENV
Light pen vertical position.		
\$0236-\$0237	566-567	
Spare; or possible BREAK key vector.		
\$0238-\$0239	568-569	
Not used.		
\$023A	570	CDEVIC
Current I/O device number.		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$023B	571	CCOMND
Bus command code.		
\$023C-\$023D	572-573	CAUX1 CAUX2
Auxiliary registers.		
\$023E	574	TEMP
Temporary I/O data register.		
\$023F	575	ERRFLG
I/O error flag.		
\$0240	576	DFLAGS
Disk flags.		
\$0241	577	DBSECT
Disk-boot sector counter.		
\$0242-\$0243	578-579	BOOTAD
Address of disk-boot loader.		
\$0244	580	COLDST
Cold-start complete flag: 0 = end of start 1 = start not done		
\$0246	582	DSKTIM
Disk timeout register.		
\$0247-\$026E	583-622	LINBUF
40-byte physical line buffer; used when moving screen data.		
\$026F	623	GPRIOR
Priority control for display handler.		
\$0270	624	PADDLO
Paddle 0 position.		
\$0271	625	PADDL1
Paddle 1 position.		
\$0272	626	PADDL2
Paddle 2 position.		
\$0273	627	PADDL3
Paddle 3 position.		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$0274	628	PADDL4
Paddle 4 position.		
\$0275	629	PADDL5
Paddle 5 position.		
\$0276	630	PADDL6
Paddle 6 position.		
\$0277	631	PADDL7
Paddle 7 position.		
\$0278	632	STICK0
Joystick 0 position.		
\$0279	633	STICK1
Joystick 1 position.		
\$027A	634	STICK2
Joystick 2 position.		
\$027B	635	STICK3
Joystick 3 position.		
\$027C	636	PTRIG0
Paddle 0 trigger sense: 0 = pressed 1 = not pressed		
\$027D	637	PTRIG1
Paddle 1 trigger sense: 0 = pressed 1 = not pressed		
\$027E	638	PTRIG2
Paddle 2 trigger sense: 0 = pressed 1 = not pressed		
\$027F	639	PTRIG3
Paddle 3 trigger sense: 0 = pressed 1 = not pressed		
\$0280	640	PTRIG4
Paddle 4 trigger sense: 0 = pressed 1 = not pressed		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$0281	641	PTRIG5
Paddle 5 trigger sense: 0 = pressed 1 = not pressed		
\$0282	642	PTRIG6
Paddle 6 trigger sense: 0 = pressed 1 = not pressed		
\$0283	643	PTRIG7
Paddle 7 trigger sense: 0 = pressed 1 = not pressed		
\$0284	644	STRIG0
Joystick 0 trigger sense: 0 = pressed 1 = not pressed		
\$0285	645	STRIG1
Joystick 1 trigger sense: 0 = pressed 1 = not pressed		
\$0286	646	STRIG2
Joystick 2 trigger sense: 0 = pressed 1 = not pressed		
\$0287	647	STRIG3
Joystick 3 trigger sense: 0 = pressed 1 = not pressed		
\$0289	649	WMODE
Cassette read/write flag: 0 = read \$80 (128) = write		
\$028A	650	BLIM
Number of bytes in cassette record buffer.		
\$028B-\$028F	651-655	
Unused.		
\$0290	656	TXTROW
Row address for the current split-screen cursor position.		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$0291-\$0292	657-658	TXTCOL	Column address for the current split-screen cursor position.
\$0293	659	TINDEX	Screen mode number for split-screen operations.
\$0294-\$0295	660-661	TXTMSC	Display address of upper-left corner of text window during split-screen operations.
\$0296-\$029B	662-667	TXTOLD	Split-screen, previous cursor information: row, column, character, and display address.
\$029C	668		Temporary-storage register.
\$029D	669	HOLD3	Temporary-storage register.
\$029E	670	SUBTMP	Temporary-storage register.
\$029F	671	HOLD2	Temporary-storage register.
\$02A0	672	DMASK	Data mask for locating graphics pixels; affected by current screen mode.
\$02A1	673	TMP LBT	Temporary register for bit masks.
\$02A2	674	ESCFLG	Escape-key flag; set to \$80 (128) by ESC keystroke; returned by 0 at subsequent keystroke.
\$02A3-\$02B1	675-689	TABMAP	Cursor tab settings for the 15 logical lines on the screen.
\$02B2-\$02B5	690-693	LOGMAP	Beginning screen-line number for each of the 15 logical lines.
\$02B6	694	INVFLG	Inverse-video flag: 0 = normal video \$80 (128) = inverse video

Chart 8-3—cont. Operating-System RAM: 512-1151

\$02B7	695	FILFLG
FILL/DRAW command flag: 0 = DRAW \$FF (255) = FILL		
\$02B8-\$02BA	696-698	TMPROW TMPCOL
Temporary row and column location registers.		
\$02BB	699	SCRFLG
Number of lines deleted from the top of the screen after a scrolling operation (minus 1).		
\$02BC	700	HOLD4
Temporary register for ATACHR during a FILL operation.		
\$02BE	702	SHFLOK
Shift/control lock mode: \$00 (0) = normal mode \$40 (64) = caps lock \$80 (128) = control lock		
\$02BF	703	BOTSCR
Number of lines available for text. Has \$18 (24) for GR.0 and \$04 (4) for split-screen modes.		
\$02C0	704	PCOLRO
Color for player and missile 0. Also color for duplicate in \$D012 (53266).		
\$02C1	705	PCOLR1
Color for player and missile 1. Also color for duplicate in \$D013 (53267).		
\$02C2	706	PCOLR2
Color for player and missile 2. Also color for duplicate in \$D014 (53268).		
\$02C3	707	PCOLR3
Color for player and missile 3. Also color for duplicate in \$D015 (53269).		
\$02C4	708	COLRO
Color of playfield 0. Also used in \$D016 (53270).		
\$02C5	709	COLR1
Color of playfield 1. Also used in \$D017 (53271).		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$02C6	710	COLR2
Color of playfield 2. Also used in \$D018 (53272).		
\$02C7	711	COLR3
Color of playfield 3. Also used in \$D019 (53273).		
\$02C8	712	COLR4
Color of playfield 4. Also used in \$D01A (53274).		
\$02E4	740	RAMSIZ
Page number of highest available RAM address.		
\$02E5-\$02E6	741-742	MEMTOP
Highest RAM address available to user.		
\$02E7-\$02E8	743-744	MEMLO
Lowest RAM address available to user.		
\$02EA-\$02ED	746-749	DVSTAT
Device status registers.		
\$02EE-\$02EF	750-751	CHBAUDL CHBAUDH
Cassette baud rate (600 baud); uses \$05CC (14846).		
\$02F0	752	CRSINH
Cursor inhibit flag: 0 = cursor ON 1 = cursor OFF		
\$02F1	753	KEYDEL
Keyboard debounce timer.		
\$02F2	754	CH1
Previous keyboard character code.		
\$02F3	755	CHACT
Character-mode/control register.		
\$02F4	756	CHBAS
Character address base.		
\$02FA	762	CHAR
Code for the most recent character at input or to output.		
\$02FB	763	ATACHR
Character code for the last-read or last-written ASCII character or plot point.		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$02FC	764	CH
Character code for the most recently accepted keystroke; set to \$FF (256) when keyboard handler accepts the data.		
\$02FD	765	FILDAT
Fill color data; used with the FILL command.		
\$02FE	766	DSPFLG
Display flag for ASCII control characters: 0 = normal control NOT 0 = character displayed		
\$02FF	767	SSFLAG
Start/stop flag for screen output: 0 = don't stop \$FF (255) = stop		
\$0300	768	DDEVIC
Serial device ID number.		
\$0301	769	DUNIT
Device unit number.		
\$0302	770	DCOMND
Device command.		
\$0303	771	DSTATS
Device status code.		
\$0304-\$0305	772-773	DBUFLO DBUFHI
Device buffer address.		
\$0306	774	DTIMLO
Device time-out value.		
\$0308-\$0309	776-777	DBYTLO DBYTHI
Device-transfer byte counter.		
\$030A-\$030B	778-779	DAUX1 DAUX2
Auxiliary device-transfer registers.		
\$030C-\$030D	780-781	TIMER1
Starting baud-rate timer reference value.		
\$030E	782	ADDCOR
Addition correction flag for baud-rate timer values.		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$030F	783	CASFLG
Cassette flag: 0 = standard serial I/O NOT 0 = cassette I/O		
\$0310-\$0311	784-785	TIMER2
Final baud-rate timer reference value.		
\$0312-\$0313	786-787	TEMP1
2-byte temporary register for baud-rate timer operations.		
\$0315	789	TEMP3
1-byte temporary register; probably used during baud-rate timer operations.		
\$0316	790	SAVIO
Serial input data detect register.		
\$0317	791	TIMFLG
Cassette baud-rate timeout (error) flag: 1 = Ok 0 = error		
\$0318	792	STACKP
Operating-system version of the 6502 stack pointer.		
\$0319	793	TSTAT
Next status byte upon return from a WAIT routine.		
\$031A-\$033F	794-831	HATABS
Handler device table (38 bytes)—device codes and handler addresses.		
\$0340	832	ICHID [0]
Device handler ID for control block 0.		
\$0341	833	ICDNO [0]
Device number for control block 0.		
\$0342	834	ICCOM [0]
Current execution command for control block 0.		
\$0343	835	ICSTA [0]
Most recent status returned by device for control block 0.		
\$0344-\$0345	836-837	ICBAL,H [0]
Buffer address for data transfer under control block 0.		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$0346-\$0347	838-839	ICPTL,H [0]	Address of PUT-a-byte routine for control block 0.
\$0348-\$0349	840-841	ICBLL,H [0]	Current length of buffer being serviced by control block 0.
\$034A-\$034B	842-843	ICAX1,2 [0]	Auxiliary registers for control block 0.
\$034C-\$034F	844-847	ISCPR [0]	Spare registers for use of handler serviced by control block 0.
\$0350	848	ICHID [1]	Device handler ID for control block 1.
\$0351	849	ICDNO [1]	Device number for control block 1.
\$0352	850	ICCOM [1]	Current execution command for control block 1.
\$0353	851	ICSTA [1]	Most recent status returned by device for control block 1.
\$0354-\$0355	852-853	ICBAL,H [1]	Buffer address for data transfer under control block 1.
\$0356-\$0357	854-855	ICPTL,H [1]	Address of PUT-a-byte routine for control block 1.
\$0358-\$0359	856-857	ICBLL,H [1]	Current length of buffer being serviced by control block 1.
\$035A-\$035B	858-859	ICAX1,2 [1]	Auxiliary registers for control block 1.
\$035C-\$035F	860-863	ISCPR [1]	Spare registers for use of handler serviced by control block 1.
\$0360	864	ICHID [2]	Device handler ID for control block 2.
\$0361	865	ICDNO [2]	Device number for control block 2.
\$0362	866	ICCOM [2]	Current execution command for control block 2.

Chart 8-3—cont. Operating-System RAM: 512-1151

\$0363	867	ICSTA [2]	Most recent status returned by device for control block 2.
\$0364-\$0365	868-869	ICBAL,H [2]	Buffer address for data transfer under control block 2.
\$0366-\$0367	870-871	ICPTL,H [2]	Address of PUT-a-byte routine for control block 2.
\$0368-\$0369	872-873	ICBLL,H [2]	Current length of buffer being serviced by control block 2.
\$036A-\$036B	874-875	ICAX1,2 [2]	Auxiliary registers for control block 2.
\$036C-\$036F	876-879	ISCPR [2]	Spare registers for use of handler serviced by control block 2.
\$0370	880	ICHID [3]	Device handler ID for control block 3.
\$0371	881	ICDNO [3]	Device number for control block 3.
\$0372	882	ICCOM [3]	Current execution command for control block 3.
\$0373	883	ICSTA [3]	Most recent status returned by device for control block 3.
\$0374-\$0375	884-885	ICBAL,H [3]	Buffer address for data transfer under control block 3.
\$0376-\$0377	886-887	ICPTL,H [3]	Address of PUT-a-byte routine for control block 3.
\$0378-\$0379	888-889	ICBLL,H [3]	Current length of buffer being serviced by control block 3.
\$037A-\$037B	890-891	ICAX1,2 [3]	Auxiliary registers for control block 3.
\$037C-\$037F	892-895	ISCPR [3]	Spare registers for use of handler serviced by control block 3.
\$0380	896	ICHID [4]	Device handler ID for control block 4.

Chart 8-3—cont. Operating-System RAM: 512-1151

\$0381	897	ICDNO [4]
Device number for control block 4.		
\$0382	898	ICCOM [4]
Current execution command for control block 4.		
\$0383	899	ICSTA [4]
Most recent status returned by device for control block 4.		
\$0384-\$0385	900-901	ICBAL,H [4]
Buffer address for data transfer under control block 4.		
\$0386-\$0387	902-903	ICPTL,H [4]
Address of PUT-a-byte routine for control block 4.		
\$0388-\$0389	904-905	ICBLL,H [4]
Current length of buffer being serviced by control block 4.		
\$038A-\$038B	906-907	ICAX1,2 [4]
Auxiliary registers for control block 4.		
\$038C-\$038F	908-911	ISCPR [4]
Spare registers for use of handler serviced by control block 4.		
\$0390	912	ICHID [5]
Device handler ID for control block 5.		
\$0391	913	ICDNO [5]
Device number for control block 5.		
\$0392	914	ICCOM [5]
Current execution command for control block 5.		
\$0393	915	ICSTA [5]
Most recent status returned by device for control block 5.		
\$0394-\$0395	916-917	ICBAL,H [5]
Buffer address for data transfer under control block 5.		
\$0396-\$0397	918-919	ICPTL,H [5]
Address of PUT-a-byte routine for control block 5.		
\$0398-\$0399	920-921	ICBLL,H [5]
Current length of buffer being serviced by control block 5.		
\$039A-\$039B	922-923	ICAX1,2 [5]
Auxiliary registers for control block 5.		

Chart 8-3—cont. Operating-System RAM: 512-1151

\$039C-\$039F	924-927	ISCPR [5]	Spare registers for use of handler serviced by control block 5.
\$03A0	928	ICHID [6]	Device handler ID for control block 6.
\$03A1	929	ICDNO [6]	Device number for control block 6.
\$03A2	930	ICCOM [6]	Current execution command for control block 6.
\$03A3	931	ICSTA [6]	Most recent status returned by device for control block 6.
\$03A4-\$03A5	932-933	ICBAL,H [6]	Buffer address for data transfer under control block 6.
\$03A6-\$03A7	934-935	ICPTL,H [6]	Address of PUT-a-byte routine for control block 6.
\$03A8-\$03A9	936-937	ICBLL,H [6]	Current length of buffer being serviced by control block 6.
\$03AA-\$03AB	938-939	ICAX1,2 [6]	Auxiliary registers for control block 6.
\$03AC-\$03AF	940-943	ISCPR [6]	Spare registers for use of handler serviced by control block 6.
\$03B0	944	ICHID [7]	Device handler ID for control block 7.
\$03B1	945	ICDNO [7]	Device number for control block 7.
\$03B2	946	ICCOM [7]	Current execution command for control block 7.
\$03B3	947	ICSTA [7]	Most recent status returned by device for control block 7.
\$03B4-\$03B5	948-949	ICBAL,H [7]	Buffer address for data transfer under control block 7.
\$03B6-\$03B7	950-951	ICPTL,H [7]	Address of PUT-a-byte routine for control block 7.

Chart 8-3—cont. Operating-System RAM: 512-1151

\$03B8-\$03B9	952-953	ICBLL,H [7]
Current length of buffer being serviced by control block 7.		
\$03BA-\$03BB	954-955	ICAX1,2 [7]
Auxiliary registers for control block 7.		
\$03BC-\$03BF	956-959	ISCPR [7]
Spare registers for use of handler serviced by control block 7.		
\$03C0-\$03E7	960-999	PRNBUF
40-byte printer buffer.		
\$03FD-\$047F	1021-1151	CASBUF
131-byte cassette buffer.		

BASIC System RAM: 1152-1535

BASIC uses some specified sections of lower RAM for its purposes. Chart 8-4 outlines those special addresses and functions for you.

Chart 8-4. BASIC Usage of System RAM: 1152-1791

\$0480-\$057D	1152-1405
BASIC syntax stack; 254 bytes.	
\$057E-\$05FF	1406-1535
INPUT buffer; 130 bytes.	
\$0600-\$06FF	1536-1791
User RAM space that is protected from everything but an INPUT buffer overflow.	

Notice the 256-byte RAM area immediately above BASIC's input line buffer. Generally speaking, this area (addresses 1536 through 1791) is protected from the operating system, BASIC and, indeed, DOS as well. It is thus a fine place to locate machine-coded programs. The only thing that can go wrong is loading a response to an INPUT statement that is more than 130 bytes long. Whenever that happens, the INPUT buffer area extends into this otherwise "protected" RAM space. Just avoid INPUTting such large amounts of information at one time.

DOS RAM USAGE: 1792-10879

If DOS is not active in the ATARI system, BASIC programming information will normally begin at address 1792. Booting DOS, however, fills at least 9088 bytes of system RAM with DOS and file-management programming. As indicated in Chart 8-5, that programming begins at address 1792 and extends to 10879, or to the current LOMEM setting. That being the case, BASIC programming picks up from the end of the DOS area.

Chart 8-5. DOS Usage of System RAM: 1792-10879, or LOMEM

\$0700-\$12FF	1792-4863
File-management RAM area; 3072 bytes.	
\$1300-\$267F	4864-9855
DOS operating system RAM area; 4992 bytes.	
\$2680-\$2A7F	9856-10879
DOS I/O buffers; 1024 bytes.	

BASIC ROM AREA: 40960-49151

The ATARI BASIC cartridge overlays any RAM that might be present in the area that it uses. Chart 8-6 offers a map of that section of the memory map.

Chart 8-6. BASIC ROM AREA: 40960-49151

\$A000-\$A04C	40960-41036
BASIC cold start routine.	
\$A04D-\$A05F	41037-41055
BASIC warm start routine.	

Chart 8-6—cont. BASIC ROM Area: 40960-49151

\$A060-\$A461	41056-42081	Syntax checking routine.
\$A462-\$A4AE	42082-42158	Search routines.
\$A4AF-\$A60C	42159-42508	Statement name table, with statement token list beginning at \$A4B1 (42161).
\$AC0D-\$A87E	42509-43134	Operator symbol table, with the operator token list beginning at \$A7E3 (42979).
\$A87F-\$A95E	43135-43358	Memory manager routines.
\$A95F-\$A9FF	43359-43519	CONT statement routine.
\$AA00-\$AA6F	43520-43631	Statement table.
\$AA70-\$AADF	43632-43743	Operator table.
\$AAE0-\$AC3E	43744-44094	Routines for executing BASIC expressions.
\$AC3F-\$AC83	44095-44163	Operator precedence routine.
\$AC84-\$AFC9	44164-45001	Operator-execution routines.
\$AFCA-\$B108	45002-45320	Function-execution routines.
\$B109-\$B817	45321-47127	Statement-execution routines.
\$B818-\$B915	47128-47381	Subroutines for CONT statement.

Chart 8-6—cont. BASIC ROM Area: 40960-49151

\$B916-\$B9B6	47382-47542	Error-handling routines.
\$B9B7-\$BA74	47543-47732	Graphics-handling routines.
\$BA75-\$BDA4	47733-48548	I/O-handling routines.
\$BDA5-\$BFF9	48549-49145	Trigonometric function routines.
\$BFFA-\$BFFB	49146-49147	Left cartridge start address.
\$BFFC	49148	Left-cartridge installed byte: 0 = cartridge installed nonzero = cartridge not installed
\$BFFD	49149	Purpose is unclear.
\$BFFE-\$BFFF	49150-49151	Cartridge initialization address; jump point after executing SYSTEM reset under BASIC.

HARDWARE I/O ROM AREA: 53248-55295

Chart 8-7 gives an overview of the hardware I/O ROM area. This is regarded as ROM only in a casual sense. Some addresses are devoted to RAM-like read/write operations; the practical difficulty, from a programmer's point of view, is that such areas are available for write operations only during the screen's vertical retrace interval.

Chart 8-7. Hardware I/O ROM Area: 53248-55295

\$D000-\$D0FF	53248-53503	CTIA (or GTIA)
\$D100-\$D1FF	53504-53759	Unused
\$D200-\$D2FF	53760-54015	POKEY
\$D300-\$D3FF	54016-54271	PIA
\$D400-\$D5FF	54272-54783	ANTIC
\$D600-\$D7FF	54784-55295	Unused

The CTIA (or GTIA) Device

CTIA, or GTIA, is a piece of system hardware—a proprietary integrated circuit, in fact—that is accessed from addresses 53248–53503 (see Chart 8-8). The chip is largely responsible for processing the ATARI's video information.

Chart 8-8. CTIA (or GTIA)
I/O Map Detail: 53248–53503

\$D000	53248	HPOSP0/MOPF	Player 0 horizontal position/missile 0 playfield collision.
\$D001	53249	HPOSP1/M1PF	Player 1 horizontal position/missile 1 playfield collision.
\$D002	53250	HPOSP2/M2PF	Player 2 horizontal position/missile 2 playfield collision.
\$D003	53251	HPOSP3/M3PF	Player 3 horizontal position/missile 3 playfield collision.
\$D004	53252	HPOSM0/POPF	Missile 0 horizontal position/player 0 playfield collision.
\$D005	53253	HPOSM1/P1PF	Missile 1 horizontal position/player 1 playfield collision.
\$D006	53254	HPOSM2/P2PF	Missile 2 horizontal position/player 2 playfield collision.
\$D007	53255	HPOSM3/P3PF	Missile 3 horizontal position/player 3 playfield collision.
\$D008	53256	SIZEP0/MOPL	Size of player 0/missile 0 player collision.
\$D009	53257	SIZEP1/M1PL	Size of player 1/missile 1 player collision.
\$D00A	53258	SIZEP2/M2PL	Size of player 2/missile 2 player collision.
\$D00B	53259	SIZEP3/M3PL	Size of player 3/missile 3 player collision.
\$D00C	53260	SIZEM/POPL	Size of all missiles/player 0 player collision.

**Chart 8-8—cont. CTIA (or GTIA)
I/O Map Detail: 53248-53503**

\$D00D	53261	GRAFP0/P1PL
Shape of player 0/player 1 player collision.		
\$D00E	53262	GRAFP1/P2PL
Shape of player 1/player 2 player collision.		
\$D00F	53263	GRAFP2/P3PL
Shape of player 2/player 3 player collision.		
\$D010	53264	GRFPF3/TRIGO
Shape of player 3/joystick 0 trigger.		
\$D011	53265	GRAFM/TRIG1
Shape of all missiles/joystick 1 trigger.		
\$D012	53266	COLOPM0/TRIG2
Color of player and missile 0/joystick 2 trigger.		
\$D013	53267	COLOPM1/TRIG3
Color of player and missile 1/joystick 3 trigger.		
\$D014	53268	COLOPM2/PAL
Color of player and missile 2/European TV sync. flag.		
\$D015	53269	COLOPM3
Color of player and missile 3.		
\$D016	53270	COLPF0
Color of playfield 0.		
\$D017	53271	COLPF1
Color of playfield 1.		
\$D018	53272	COLPF2
Color of playfield 2.		
\$D019	53273	COLPF3
Color of playfield 3.		
\$D01A	53274	COLBK
Color of background.		
\$D01B	53275	PRIOR
Figure priority-select; determines whether a figure is to be located behind or in front of another.		

**Chart 8-8—cont. CTIA (or GTIA)
I/O Map Detail: 53248-53503**

\$D01C	53276	VDLAY
Vertical delay; used for moving players or missiles in their 1-line or 2-line resolution formats.		
\$D01D	53277	GRACTL
Turns trigger, missile and player elements on and off.		
\$D01E	53278	HITCLR
Flag register for clearing collision events.		
\$D01F	53279	CONSOL
Console-button register.		
\$D020-\$D0FF	53280-53503	
Duplicate of addresses \$D000-\$D01F (53248-53279).		

The POKEY Device

Like CTIA, POKEY is a piece of system hardware. It handles several different kinds of digital tasks, including generating the audio tones, random noise and numbers, strobing the keyboard, polling the game ports, and generating timing pulses.

Chart 8-9 summarizes some of the important addresses for the POKEY chip.

Chart 8-9. POKEY Map Detail: 53760-54015

\$D200	53760	AUDF1/POT0
Audio voice 1 frequency/paddle 0 setting.		
\$D201	53761	AUDC1/POT1
Audio voice 1 volume and distortion/paddle 1 setting.		
\$D202	53762	AUDF2/POT2
Audio voice 2 frequency/paddle 2 setting.		
\$D203	53763	AUDC2/POT3
Audio voice 2 volume and distortion/paddle 3 setting.		

Chart 8-9—cont. POKEY Map Detail: 53760-54015

\$D204	53764	AUDF3/POT4
Audio voice 3 frequency/paddle 2 setting.		
\$D205	53765	AUDC3/POT5
Audio voice 3 volume and distortion/paddle 5 setting.		
\$D206	53766	AUDF4/POT6
Audio voice 4 frequency/paddle 6 setting.		
\$D207	53767	AUDC4/POT7
Audio voice 4 volume and distortion/paddle 7 setting.		
\$D208	53768	AUDCTL/ALLPOT
Audio voice master control/all paddles.		
\$D209	53769	STIMER/KBCODE
Start POKEY timers/keyboard latch.		
\$D20A	53770	RANDOM
Random-number counter/register.		
\$D20B	53771	POTGO
Read paddles flag.		
\$D20C	53772	
Not used.		
\$D20D	53773	SEROUT/SERIN
Serial I/O register.		
\$D20E	53774	IRQEN/IRQST
Interrupt request enable and interrupt status request register.		
\$D20F	53775	SKCTL/SKSTAT
Serial control and status register; includes keyboard debounce, keyboard scanning, and serial port mode control.		
\$D210-\$D2FF	53776-54015	
Duplicate of \$D200-\$D20F (53760-53775)		

The PIA Device

The Peripheral Interface Adaptor (PIA) hardware system is accessible from addresses 54016 through 54271. Chart 8-10 outlines the functions performed at some of the critical address locations.

Chart 8-10. PIA Map Detail: 54016-54271

\$D300	54016	PORTA
Register for controller jacks 1 and 2.		
\$D301	54017	PORTB
Register for controller jacks 3 and 4.		
\$D302	54018	PACTL
Port A control register.		
\$D303	54019	PBCTL
Port B control register.		
\$D304-\$D3FF	54020-54271	
Duplicates of \$D300-\$D303 (54016-54019)		

The ANTIC Device

ANTIC is a hardware device that is largely responsible for the ATARI system's unique graphics capabilities. Chart 8-11 lists the important addresses for that chip.

Chart 8-11. ANTIC Map Detail: 54272-54303

\$D400	54272	DMACTL
Direct-memory-access control (DMA).		
\$D401	54273	CHACTL
Character mode control.		
\$D402-\$D403	54274-54275	DLISTL DLISTH
Display list pointer.		
\$D404	54276	HSCROL
Horizontal-scroll enable.		

Chart 8-11—cont. ANTIC Map Detail: 54272-54303

\$D405	54277	VSCROL
Vertical-scroll enable.		
\$D406	54278	
Unused.		
\$D407	54279	PMBASE
Most-significant byte of the player/missile base address.		
\$D408	54280	
Unused.		
\$D409	54281	CHBASE
Text character bit map base address.		
\$D40A	54282	WSYNCH
Wait-for-horizontal sync flag.		
\$D40B	54283	VCOUNT
Vertical-scan line counter.		
\$D40C	54284	PENH
Horizontal position of light pen.		
\$D40D	54285	PENV
Vertical position of light pen.		
\$D40E	54286	NMIEN
Enable nonmaskable interrupt.		
\$D40F	54287	NMIREN/NMIST
Clear interrupt-request, reset any nonmaskable interrupts, and return current interrupt status.		
\$D410-\$D41F	54288-54303	
Duplicate of \$D400-\$D40F 54272-54287		

OPERATING SYSTEM ROM AREA: 55296-65535

The uppermost section of the ATARI memory map is devoted to ROM operations for the operating system, itself. This section is common to both the 800 and 400 systems, and it is used whether a BASIC cartridge is installed or not.

Chart 8-12 outlines the major functions and entry-point addresses.

**Chart 8-12. Operating System
ROM Area: 55296-65535**

\$D800-\$DFFF	55296-57393	Floating-point arithmetic.
\$E000-\$E3FF	57344-58367	Internal character set (see Appendix C2).
\$E400-\$E40F	58368-58383	Editor vectors.
\$E410-\$E41F	58384-58399	Screen vectors.
\$E420-\$E42F	58400-58415	Keyboard vectors.
\$E430-\$E43F	58416-58431	Printer vectors.
\$E440-\$E44F	58432-58447	Cassette vectors.
\$E450-\$E4A5	58448-58533	Jump and initial RAM vectors.
\$EA6-\$E6D4	58534-59092	Central I/O (CIO) handler addresses.
\$E6D5-\$E943	59093-59715	Interrupt handler addresses.
\$E944-\$EDE9	59716-60905	Serial I/O (SIO) routines.
\$EDEA-\$EE77	60906-61047	Disk handler routines.
\$EE78-\$EF40	61048-61248	Printer handler routines.
\$EF41-\$F0E2	61249-61666	Cassette handler routines.

**Chart 8-12—cont. Operating System
ROM Area: 55296-65535**

\$F0E3-\$F3E3	61667-62435
Monitor handler routines.	
\$F3E4-\$FFFF	62436-65535
Display and keyboard handler routines.	

Chapter 9

The 6502 Instruction Set

The hallmarks of machine-language programs are higher operating speeds and more efficient use of available RAM space. There are two different ways to work with machine-language programs with the ATARI system and, indeed, most personal computers.

One technique uses a BASIC program to POKE decimal versions of the machine instructions and data into a specified area of memory (also specified in a decimal format), and then execute the routines from BASIC by means of the USR command.

An alternative approach is to take advantage of the ATARI Assembler Editor software. The programmer can write, enter, edit, and debug machine-language programs in a source code, or assembly language, format. Although source programs generally use a hexadecimal data and addressing format, the Assembler Editor package offers the option of working with decimal values as well.

It is beyond the scope of this book to deal with programming procedures for the 6502 microprocessor. Any good book on the subject of 6502 programming or, indeed, ATARI's *Assembler Editor Manual* can be of valuable help in this regard.

Table 9-1 summarizes the 6502 instruction set, showing the op codes in both decimal and hexadecimal forms.

**Table 9-1. Summary of the 6502 Instruction
Set as Organized by Op Codes**

Hex Op Code	Dec Op Code	Mnemonic	
\$00	0	BRK	implied
\$01	1	ORA	(indirect, x)
\$02	2		not used
\$03	3		not used
\$04	4		not used
\$05	5	ORA	zero page
\$06	6	ASL	zero page
\$07	7		not used
\$08	8	PHP	implied
\$09	9	ORA	immediate
\$0A	10	ASL	accumulator
\$0B	11		not used
\$0C	12		not used
\$0D	13	ORA	absolute
\$0E	14	ASL	absolute
\$0F	15		not used
\$10	16	BPL	relative
\$11	17	ORA	(indirect),Y
\$12	18		not used
\$13	19		not used
\$14	20	JSR	
\$15	21	ORA	zero page,X
\$16	22	ASL	zero page,X
\$17	23		not used
\$18	24	CLC	implied
\$19	25	ORA	absolute,Y
\$1A	26		not used
\$1B	27		not used
\$1C	28		not used
\$1D	29	ORA	absolute,X
\$1E	30	ASL	absolute,X
\$1F	31		not used

Table 9-1—cont. Summary of the 6502 Instruction Set as Organized by Op Codes

Hex Op Code	Dec Op Code	Mnemonic	
\$20	32	JSR	absolute
\$21	33	AND	(indirect,X)
\$22	34		not used
\$23	35		not used
\$24	36	BIT	zero page
\$25	37	AND	zero page
\$26	38	ROL	zero page
\$27	39		not used
\$28	40	PLP	implied
\$29	41	AND	immediate
\$2A	42	ROL	accumulator
\$2B	43		not used
\$2C	44	BIT	absolute
\$2D	45	AND	absolute
\$2E	46	ROL	absolute
\$2F	47		not used
\$30	48	BMI	relative
\$31	49	AND	(indirect),Y
\$32	50		not used
\$33	51		not used
\$34	52		not used
\$35	53	AND	zero page,X
\$36	54	ROL	zero page,X
\$37	55		not used
\$38	56	SEC	implied
\$39	57	AND	absolute,Y
\$3A	58		not used
\$3B	59		not used
\$3C	60		not used
\$3D	61	AND	absolute,X
\$3E	62	ROL	absolute,X
\$3F	63		not used

Table 9-1—cont. Summary of the 6502 Instruction Set as Organized by Op Codes

Hex Op Code	Dec Op Code	Mnemonic	
\$40	64	RTI	implied
\$41	65	EOR	(indirect,X)
\$42	66		not used
\$43	67		not used
\$44	68		not used
\$45	69	EOR	zero page
\$46	70	LSR	zero page
\$47	71		not used
\$48	72	PHA	implied
\$49	73	EOR	immediate
\$4A	74	LSR	accumulator
\$4B	75		not used
\$4C	76	JMP	absolute
\$4D	77	EOR	absolute
\$4E	78	LSR	absolute
\$4F	79		not used
\$50	80	BVC	relative
\$51	81	EOR	(indirect),Y
\$52	82		not used
\$53	83		not used
\$54	84		not used
\$55	85	EOR	zero page,X
\$56	86	LSR	zero page,X
\$57	87		not used
\$58	88	CLI	implied
\$59	89	EOR	absolute,Y
\$5A	90		not used
\$5B	91		not used
\$5C	92		not used
\$5D	93	EOR	absolute,X
\$5E	94	LSR	absolute,X
\$5F	95		not used

Table 9-1—cont. Summary of the 6502 Instruction Set as Organized by Op Codes

Hex Op Code	Dec Op Code	Mnemonic	
\$60	96	RTS	implied
\$61	97	ADC	(indirect,X)
\$62	98		not used
\$63	99		not used
\$64	100		not used
\$65	101	ADC	zero page
\$66	102	ROR	zero page
\$67	103		not used
\$68	104	PLA	implied
\$69	105	ADC	immediate
\$6A	106	ROR	accumulator
\$6B	107		not used
\$6C	108	JMP	indirect
\$6D	109	ADC	absolute
\$6E	110	ROR	absolute
\$6F	111		not used
\$70	112	BVS	relative
\$71	113	ADC	(indirect),Y
\$72	114		not used
\$73	115		not used
\$74	116		not used
\$75	117	ADC	zero page,X
\$76	118	ROR	zero page,X
\$77	119		not used
\$78	120	SEI	implied
\$79	121	ADC	absolute,Y
\$7A	122		not used
\$7B	123		not used
\$7C	124		not used
\$7D	125	ADC	absolute,X
\$7E	126	ROR	absolute,X
\$7F	127		not used

Table 9-1—cont. Summary of the 6502 Instruction Set as Organized by Op Codes

Hex Op Code	Dec Op Code	Mnemonic	
\$80	128	BCS	relative
\$81	129	STA	(indirect,X)
\$82	130		not used
\$83	131		not used
\$84	132	STY	zero page
\$85	133	STA	zero page
\$86	134	STX	zero page
\$87	135		not used
\$88	136	DEY	implied
\$89	137		not used
\$8A	138	TXA	implied
\$8B	139		not used
\$8C	140	STY	absolute
\$8D	141	STA	absolute
\$8E	142	STX	absolute
\$8F	143		not used
\$90	144	BCC	relative
\$91	145	STA	(indirect),Y
\$92	146		not used
\$93	147		not used
\$94	148	STY	zero page,X
\$95	149	STA	zero page,X
\$96	150	STX	zero page,Y
\$97	151		not used
\$98	152	TYA	implied
\$99	153	STA	absolute,Y
\$9A	154	TXS	implied
\$9B	155		not used
\$9C	156		not used
\$9D	157	STA	absolute,X
\$9E	158		not used
\$9F	159		not used

Table 9-1—cont. Summary of the 6502 Instruction Set as Organized by Op Codes

Hex Op Code	Dec Op Code	Mnemonic	
\$A0	160	LDY	immediate
\$A1	161	LDA	(indirect,X)
\$A2	162	LDX	immediate
\$A3	163		not used
\$A4	164	LDY	zero page
\$A5	165	LDA	zero page
\$A6	166	LDX	zero page
\$A7	167		not used
\$A8	168	TAY	implied
\$A9	169	LDA	immediate
\$AA	170	TAX	implied
\$AB	171		not used
\$AC	172	LDY	absolute
\$AD	173	LDA	absolute
\$AE	174	LDX	absolute
\$AF	175		not used
\$B0	176	BCS	
\$B1	177	LDA	(indirect),Y
\$B2	178		not used
\$B3	179		not used
\$B4	180	LDY	zero page,X
\$B5	181	LDA	zero page,X
\$B6	182	LDX	zero page,Y
\$B7	183		not used
\$B8	184	CLV	implied
\$B9	185	LDA	absolute,Y
\$BA	186	TSX	implied
\$BB	187		not used
\$BC	188	LDY	absolute,X
\$BD	189	LDA	absolute,X
\$BE	190	LDX	absolute,Y
\$BF	191		not used

Table 9-1—cont. Summary of the 6502 Instruction Set as Organized by Op Codes

Hex Op Code	Dec Op Code	Mnemonic	
\$C0	192	CPY	immediate
\$C1	193	CMP	(indirect,X)
\$C2	194		not used
\$C3	195		not used
\$C4	196	CPY	zero page
\$C5	197	CMP	zero page
\$C6	198	DEC	zero page
\$C7	199		not used
\$C8	200	INY	implied
\$C9	201	CMP	immediate
\$CA	202	DEX	implied
\$CB	203		not used
\$CC	204	CPY	absolute
\$CD	205	CMP	absolute
\$CE	206	DEC	absolute
\$CF	207		not used
\$D0	208	BNE	relative
\$D1	209	CMP	(indirect),Y
\$D2	210		not used
\$D3	211		not used
\$D4	212		not used
\$D5	213	CMP	zero page,X
\$D6	214	DEC	zero page,X
\$D7	215		not used
\$D8	216	CLD	implied
\$D9	217	CMP	absolute,Y
\$DA	218		not used
\$DB	219		not used
\$DC	220		not used
\$DD	221	CMP	absolute,X
\$DE	222	DEC	absolute,X
\$DF	223		not used

Table 9-1—cont. Summary of the 6502 Instruction Set as Organized by Op Codes

Hex Op Code	Dec Op Code	Mnemonic	
\$E0	224	CPX	immediate
\$E1	225	SBC	(indirect,X)
\$E2	226		not used
\$E3	227		not used
\$E4	228	CPX	zero page
\$E5	229	SBC	zero page
\$E6	230	INC	zero page
\$E7	231		not used
\$E8	232	INX	implied
\$E9	233	SBC	immediate
\$EA	234	NOP	implied
\$EB	235		not used
\$EC	236	CPX	absolute
\$ED	237	SBC	absolute
\$EE	238	INC	absolute
\$EF	239		not used
\$F0	240	BEQ	relative
\$F1	241	SBC	(indirect),Y
\$F2	242		not used
\$F3	243		not used
\$F4	244		not used
\$F5	245	SBC	zero page,X
\$F6	246	INC	zero page,X
\$F7	247		not used
\$F8	248	SED	implied
\$F9	249	SBC	absolute,Y
\$FA	250		not used
\$FB	251		not used
\$FC	252		not used
\$FD	253	SBC	absolute,X
\$FE	254	INC	absolute,X
\$FF	255		not used

Fig. 9-1 illustrates the organization of registers within the 6502 microprocessor.

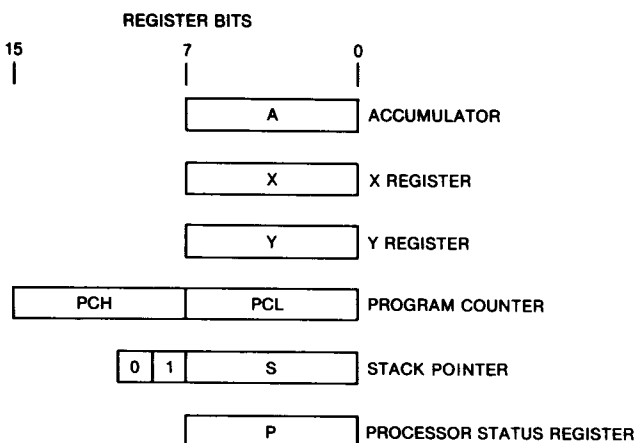


Fig. 9-1. The 6502 internal registers.

The remainder of this chapter is devoted to a detailed version of the instructions set. The material is arranged by instruction mnemonics, and in alphabetical order. Asterisks shown under the heading, *Status register*, indicate status-register bits that are affected by the operation.

ADC ADD memory to accumulator with carry

Operation: $A+M+C \rightarrow A, C$

Status register (P)

N	Z	C	I	D	V
*	*	*	-	-	*

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Immediate	ADC #Oper	69	105	2
Zero Page	ADC Oper	65	101	2
Zero Page,X	ADC Oper,X	75	117	2
Absolute	ADC Oper	6D	109	3
Absolute,X	ADC Oper,X	7D	125	3
Absolute,Y	ADC Oper,Y	79	121	3
(Indirect,X)	ADC (Oper,X)	61	97	2
(Indirect),Y	ADC (Oper),Y	71	113	2

AND Logically AND memory with accumulator

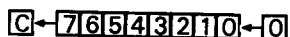
Operation: $A \wedge M \rightarrow A$

Status register (P)

N	Z	C	I	D	V
*	-	*	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Immediate	AND #Oper	29	41	2
Zero Page	AND Oper	25	37	2
Zero Page,X	AND Oper,X	35	53	2
Absolute	AND Oper	2D	45	3
Absolute,X	AND Oper,X	3D	61	3
Absolute,Y	AND Oper,Y	39	57	3
(Indirect,X)	AND (Oper,X)	21	33	2
(Indirect),Y	AND (Oper),Y	31	49	2

ASL Shift memory or accumulator 1 bit to the left
Operation:



Status register (P)

N Z C I D V
* * * - - -

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Accumulator	ASL	A	0A	10	1
Zero Page	ASL	Oper	06	6	2
Zero Page,X	ASL	Oper,X	16	22	2
Absolute	ASL	Oper	0E	14	3
Absolute,X	ASL	Oper,X	1E	30	3

BCC Branch on carry clear
Operation: Branch if C=0

Status register (P)

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Relative	BCC	Oper	90	144	2

BCS Branch on carry set
Operation: Branch on C=1

Status register (P)

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Relative	BCS	Oper	80	128	2

BEQ Branch on result zero

Operation: Branch if Z=1

Status register (P)

N	Z	C	I	D	V
-	-	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Relative	BEQ Oper	F0	240	2

BIT Test memory bits with accumulatorOperation: $A \wedge M, M_7 \leftarrow N, M_6 \leftarrow V$

Status register (P)

N	Z	C	I	D	V
M_7	*	-	-	-	M_6

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Zero Page	BIT Oper	24	36	2
Absolute	BIT Oper	2C	44	3

BMI Branch on result minus

Operation: Branch if N=1

Status register (P)

N	Z	C	I	D	V
-	-	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Relative	BMI Oper	30	48	2

BNE Branch on result not zero

Operation: Branch if Z=0

Status register (P)

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Relative	BNE Oper	D0	208	2

BPL Branch on result plus

Operation: Branch if N=0

Status register (P)

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Relative	BPL Oper	10	16	2

BRK Forced break

Operation: Interrupt, PC+2↓P↓

Status register (P)

N Z C I D V
- - - 1 - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	BRK	00	0	1

BVC Branch on overflow clear

Operation: Branch if V=0

Status register (P)

N	Z	C	I	D	V
-	-	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Relative	BVC Oper	50	80	2

BVS Branch on overflow set

Operation: Branch if V=1

Status register (P)

N	Z	C	I	D	V
-	-	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Relative	BVS Oper	70	112	2

CLC Clear carry flag

Operation: 0→C

Status register (P)

N	Z	C	I	D	V
-	-	-	0	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	CLC	18	24	1

CLD Clear decimal arithmetic mode

Operation: 0→D

Status register (P)

N	Z	C	I	D	V
-	0	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	CLD	D8	216	1

CLI Clear interrupt disable bit

Operation: 0→I

Status register (P)

N	Z	C	I	D	V
-	-	-	0	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	CLI	58	88	1

CLV Clear overflow flag

Operation: 0→V

Status register (P)

N	Z	C	I	D	V
0	-	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	CLV	B8	184	1

CMP Compare memory and accumulator

Operation: A - M

Status register (P)

N	Z	C	I	D	V
*	*	*	-	-	-

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Immediate	CMP	#Oper	C9	201	2
Zero Page	CMP	Oper	C5	197	2
Zero Page,X	CMP	Oper,X	D5	213	2
Absolute	CMP	Oper	CD	205	3
Absolute,X	CMP	Oper,X	DD	221	3
Absolute,Y	CMP	Oper,Y	D9	217	3
(Indirect,X)	CMP	(Oper,X)	C1	193	2
(Indirect),Y	CMP	(Oper),Y	D1	209	2

CPX Compare memory and X register

Operation: X - M

Status register (P)

N	Z	C	I	D	V
*	*	*	-	-	-

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Immediate	CPX	#Oper	E0	224	2
Zero Page	CPX	Oper	E4	228	2
Absolute	CPX	Oper	EC	236	3

CPY Compare memory and Y register

Operation: $Y - M$

Status register (P)

N	Z	C	I	D	V
*	*	*	-	-	-

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Immediate	CPY	#Oper	C0	192	2
Zero Page	CPY	Oper	C4	196	2
Absolute	CPY	Oper	CC	204	3

DEC Decrement memory by 1

Operation: $M-1 \rightarrow M$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Zero Page	DEC	Oper	C6	198	2
Zero Page,X	DEC	Oper,X	D6	214	2
Absolute	DEC	Oper	CE	206	3
Absolute,X	DEC	Oper,X	DE	222	3

DEX Decrement X register by 1

Operation: $X-1 \rightarrow X$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	DEX	CA	202	1

DEY Decrement Y register by 1

Operation: $Y-1 \rightarrow Y$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	DEY	88	136	1

EOR Logically EXCLUSIVE-OR memory and accumulator

Operation: $A \vee M \rightarrow A$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Immediate	EOR #Oper	49	73	2
Zero Page	EOR Oper	45	69	2
Zero Page,X	EOR Oper,X	55	85	2
Absolute	EOR Oper	4D	77	3
Absolute,X	EOR Oper,X	5D	93	3
Absolute,Y	EOR Oper,Y	59	89	3
(Indirect,X)	EOR (Oper,X)	41	65	2
(Indirect),Y	EOR (Oper),Y	51	81	2

INC Increment memory by 1

Operation: $M+1 \rightarrow M$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Zero Page	INC	Oper	E6	230	2
Zero Page,X	INC	Oper,X	F6	246	2
Absolute	INC	Oper	EE	238	3
Absolute,X	INC	Oper,X	FE	254	3

INX Increment X register by 1

Operation: $X+1 \rightarrow X$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	INX	E8	232	1

INY Increment Y register by 1

Operation: $Y+1 \rightarrow Y$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	INY	C8	200	1

JMP Unconditional jump to new address

Operation: (PC+1) → PCL
(PC+2) → PCH

Status register (P)

N Z C I D V
- - - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Absolute	JMP Oper	4C	76	3
Indirect	JMP (Oper)	6C	108	3

JSR Jump to new address and save return address

Operation: PC+2 ↓
(PC+1) → PCL
(PC+2) → PCH

Status register (P)

N Z C I D V
- - - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Absolute	JSR Oper	20	32	3

LDA Load the accumulator with memory

Operation: $M \rightarrow A$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Immediate	LDA #Oper	A9	169	2
Zero Page	LDA Oper	A5	165	2
Zero Page,X	LDA Oper,X	B5	181	2
Absolute	LDA Oper	AD	173	3
Absolute,X	LDA Oper,X	BD	189	3
Absolute,Y	LDA Oper,Y	B9	185	3
(Indirect,X)	LDA (Oper,X)	A1	161	2
(Indirect),Y	LDA (Oper),Y	B1	177	2

LDX Load X register with memory

Operation: $M \rightarrow X$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Immediate	LDX #Oper	A2	162	2
Zero Page	LDX Oper	A6	166	2
Zero Page,Y	LDX Oper,Y	B6	182	2
Absolute	LDX Oper	AE	174	3
Absolute,Y	LDX Oper,Y	BE	190	3

LDY Load Y register with memory

Operation: $M \rightarrow Y$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Immediate	LDY #Oper	A0	160	2
Zero Page	LDY Oper	A4	164	2
Zero Page,X	LDY Oper,X	B4	180	2
Absolute	LDY Oper	AC	172	3
Absolute,X	LDY Oper,X	BC	188	3

LSR Shift memory or accumulator 1 bit to right

Operation:

C ← **7 6 5 4 3 2 1 0** ← **0**

Status register (P)

N	Z	C	I	D	V
0	*	*	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Accumulator	LSR A	4A	74	1
Zero Page	LSR Oper	46	70	2
Zero Page,X	LSR Oper,X	56	86	2
Absolute	LSR Oper	4E	78	3
Absolute,X	LSR Oper,X	5E	94	3

NOP No operation

Status register (P)

N Z C I D V
- - - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	NOP	EA	234	1

ORA Logically OR memory and accumulator

Operation: $A \vee M \rightarrow A$

Status register (P)

N Z C I D V
* * - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Immediate	ORA #Oper	09	9	2
Zero Page	ORA Oper	05	5	2
Zero Page,X	ORA Oper,X	15	21	2
Absolute	ORA Oper	0D	13	3
Absolute,X	ORA Oper,X	1D	29	3
Absolute,Y	ORA Oper,Y	19	25	3
(Indirect,X)	ORA (Oper,X)	01	1	2
(Indirect),Y	ORA (Oper),Y	11	17	2

PHA Push accumulator to top of stack

Operation: $A \downarrow$

Status register (P)

N Z C I D V
- - - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	PHA	48	72	1

PHP Push status register to top of stack

Operation: P ↓

Status register (P)

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	PHP	08	8	1

PLA Pull top of stack to accumulator

Operation: A ↑

Status register (P)

N Z C I D V
* * - - - -

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	PLA	68	104	1

PLP Pull top of stack to status register

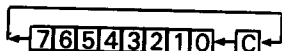
Operation: P ↑

Status register (P)

From Stack

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	PLP	28	40	1

ROL Rotate memory or accumulator 1 bit to left
Operation:

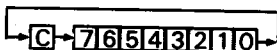


Status register (P)

N	Z	C	I	D	V
*	*	*	-	-	-

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Accumulator	ROL	A	2A	42	1
Zero Page	ROL	Oper	26	38	2
Zero Page,X	ROL	Oper,X	36	54	2
Absolute	ROL	Oper	2E	46	3
Absolute,X	ROL	Oper,X	3E	62	3

ROR Rotate memory or accumulator 1 bit to right
Operation:



Status register (P)

N	Z	C	I	D	V
*	*	*	-	-	-

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Accumulator	ROR	A	6A	106	1
Zero Page	ROR	Oper	66	102	2
Zero Page,X	ROR	Oper,X	76	118	2
Absolute	ROR	Oper	6E	110	3
Absolute,X	ROR	Oper,X	7E	126	3

RTI Return from interruptOperation: $P \uparrow PC \uparrow$

Status register (P)

From Stack

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	RTI	40	64	1

RTS Return from subroutineOperation: $PC \uparrow, PC-1 \rightarrow PC$

Status register (P)

N	Z	C	I	D	V
-	-	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	RTS	60	96	1

SBC Subtract memory from accumulator with borrowOperation: $A-M-\bar{C} \rightarrow A$

Status register (P)

N	Z	C	I	D	V
*	*	*	-	-	*

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Immediate	SBC #Oper	E9	233	2
Zero Page	SBC Oper	E5	229	2
Zero Page,X	SBC Oper,X	F5	245	2
Absolute	SBC Oper	ED	237	3
Absolute,X	SBC Oper,X	FD	253	3
Absolute,Y	SBC Oper,Y	F9	249	3
(Indirect,X)	SBC (Oper,X)	E1	225	2
(Indirect),Y	SBC (Oper),Y	F1	241	2

SEC Set carry flag
Operation: $1 \rightarrow C$

Status register (P)

N	Z	C	I	D	V
-	-	1	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	SEC	38	56	1

SED Set decimal arithmetic mode
Operation: $1 \rightarrow D$

Status register (P)

N	Z	C	I	D	V
-	-	-	-	1	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	SED	F8	248	1

SEI Set disable interrupt flag
Operation: $1 \rightarrow I$

Status register (P)

N	Z	C	I	D	V
-	-	-	1	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	SEI	78	120	1

STA Store accumulator in memory

Operation: $A \rightarrow M$

Status register (P)

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Zero Page	STA	Oper	85	133	2
Zero Page,X	STA	Oper,X	95	149	2
Absolute	STA	Oper	8D	141	3
Absolute,X	STA	Oper,X	9D	157	3
Absolute,Y	STA	Oper,Y	99	153	3
(Indirect,X)	STA	(Oper,X)	81	129	2
(Indirect),Y	STA	(Oper),Y	91	145	2

STX Store X register in memory

Operation: $X \rightarrow M$

Status register (P)

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Zero Page	STX	Oper	86	135	2
Zero Page,Y	STX	Oper,Y	96	150	2
Absolute	STX	Oper	8E	142	3

STY Store Y register in memory

Operation: $Y \rightarrow M$

Status register (P)

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Format		Hex Op Code	Dec Op Code	Bytes
Zero Page	STY	Oper	84	132	2
Zero Page,X	STY	Oper,X	94	148	2
Absolute	STY	Oper	8C	140	3

TAX Transfer accumulator to X register

Operation: $A \rightarrow X$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	TAX	AA	170	1

TAY Transfer accumulator to Y register

Operation: $A \rightarrow Y$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	TAY	A8	168	1

TSX Transfer stack pointer to X register

Operation: $S \rightarrow X$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	TSX	BA	186	1

TXA Transfer X register to accumulator

Operation: $X \rightarrow A$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	TXA	8A	138	1

TSX Transfer X register to stack pointer

Operation: $S \rightarrow X$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	TSX	BA	186	1

TYA Transfer Y register to accumulator

Operation: $Y \rightarrow A$

Status register (P)

N	Z	C	I	D	V
*	*	-	-	-	-

Addressing Mode	Assembly Language Format	Hex Op Code	Dec Op Code	Bytes
Implied	TYA	98	152	1

Appendix A

Number-System Base Conversions

Just about any computer (certainly the ATARI system) is essentially a binary machine; the 6502 microprocessor does all of its control, arithmetic, and logic operations in a base 2, or binary, number system. And it so happens that the 6502 works with 8-bit binary numbers—a full byte of them.

People do not think and work with binary numbers very well, however. Such numbers, being made up exclusively of 0s and 1s, are very cumbersome. One alternative to purely binary representations of numbers is hexadecimal numbers. The hexadecimal (base 16) number system looks at binary numbers in groups of four; every group of four binary numbers (sometimes called a nibble) can be represented by a single hexadecimal number. So, instead of having to work with strings of eight 0s and 1s in base-2 binary, it is possible to work with just two hexadecimal characters.

While, indeed, many machine-language programmers can learn to work with hexadecimal numbers with great proficiency, the general population still prefers the ordinary decimal (base 10) number system. ATARI engineers were aware of that fact, so they designed the ATARI BASIC ROM cartridge exclusively around the decimal number system.

As long as one works with ATARI BASIC in its most elementary fashion—doing no special addressing or machine-language work—there is no need to be aware of hexadecimal or binary numbers. But hexadecimal numbers become quite helpful when doing extensive machine-language programming.

Thus, programmers who are working their way deeper and deeper into the ATARI system will find themselves having to make conversions between decimal and hexadecimal numbers and, eventually, between binary and hexadecimal numbers. The purpose of this appendix is to make such conversion tasks as simple as possible.

There are many ways to approach the conversions between these three different systems; the following are the most straightforward.

HEXADECIMAL-TO-DECIMAL CONVERSIONS

In the ATARI system, data is carried as a 1-byte (two-hexadecimal-number) code. Addresses are carried as 1-byte codes for the zero-page memory and as 2-byte codes for the remainder of the usable memory space. Table A-1 can be very helpful for translating hexadecimal numbers into their decimal counterparts. This sort of situation often arises when one is writing programs in both BASIC and machine language.

The table can be used for converting up to four hexadecimal places, or nibbles, to their counterpart. Notice that there are four major columns, labeled 1 through 4. These column numbers represent the relative position of the hexadecimal characters as they are usually written, with the least-significant nibble on the right and the most-significant nibble on the left.

Table A-1. Hexadecimal/Decimal Conversion Table

MSB 4		3		2		LSB 1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

To see how the table works, suppose that you want to convert the hexadecimal value \$1A3F into decimal. The first character on the left takes a decimal equivalent shown in column 4—4096. The second character from the left takes on the value from column 3—2560. The last two figures get their decimal equivalents from columns 2 and 1—48 and 15, respectively. Then, to get the true decimal value, add those decimal equivalents: $4096 + 2560 + 48 + 15$, or 6719. In other words, \$1A3F is equal to 6719 in decimal.

If you are converting a two-place hexadecimal number, just use columns 2 and 1. Hexadecimal \$C3, for instance, is equal to $192 + 3$, or decimal 195.

Table A-1 is adequate for hexadecimal-to-decimal conversions for all the usual sort of work on the ATARI system.

DECIMAL-TO-HEXADECIMAL CONVERSIONS

When working back and forth between BASIC and machine-language routines, it is often necessary to convert decimal data and addresses into hexadecimal notation. Table A-1 comes to the rescue again. The procedure is a rather straightforward one, but it involves several steps.

Suppose, for example, that you want to convert decimal 65 into its hexadecimal counterpart. First, find the decimal number on the table that is equal to, or less than, the desired decimal number. The decimal number in this example is 65, and the closest value less than 65 is 64. The 64 is equivalent to a hexadecimal \$4 in column 2. Thus the most-significant number in the hexadecimal representation is 4.

Next, subtract that 64 from the number that you are working with: $65 - 64 = 1$. Then look up the hexadecimal value of the 1 in the next-lower column of the table—column 1 in this instance. The hexadecimal version of that number is \$1. Putting together those two hexadecimal characters, you get a \$41. Indeed, decimal 65 translates into hexadecimal \$41.

By way of a somewhat more involved conversion, suppose that you must convert decimal 19314 into hexadecimal notation.

Looking through the columns of decimal numbers in the table, you find that 16384 is the next-lower value; it translates into hexadecimal \$4 in column 4. So you are going to end up with a four-digit hexadecimal number, with the digit on the left being a 4.

To get the next-lower place value, subtract the table value 16384 from 19314: $19314 - 16384 = 2930$. The next lower decimal value in this case is 2816 from column 3; that turns up a \$B as the next hexadecimal character. So far, the number is \$4B.

Now subtract the table value 2816 from 2930: $2930 - 2816 = 114$. The next-lower decimal value from column 2 is 112, and its hexadecimal counterpart is 7. And to this point, the hexadecimal number is \$4B7.

Finally, subtract the table value 112 from 114. $114 - 112 = 2$. From column 1, decimal 2 is the same as hexadecimal \$2; so the final hexadecimal character is \$2.

Putting all this together, it turns out that decimal 19314 is the same as hexadecimal \$4B72. Fig. A-1 summarizes the operation.

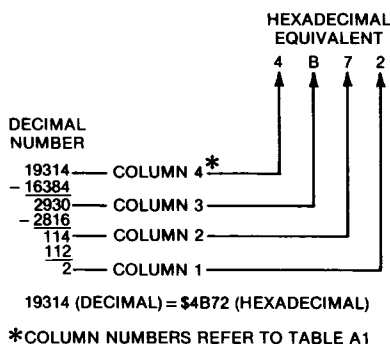


Fig. A-1. Converting decimal values to hexadecimal values.

CONVENTIONAL DECIMAL TO 2-BYTE DECIMAL FORMAT

When POKEing addresses as 2-byte numbers into memory, it is necessary to convert the address to be affected into a 2-byte format. In decimal, such an operation isn't easy, but it is all a part of setting up address locations in decimal-oriented BASIC.

By way of an example, suppose that you are to load a 2-byte version of decimal address 1234 into memory addresses 16787 and 16788. That number to be stored, 1234, is too large for either of those 1-byte addresses, so it has to be broken up into two parts: one for each of the address locations.

Before a decimal number can be divided into a 2-byte version, it must be converted into hexadecimal form. Using the decimal-to-hexadecimal conversion described in the previous section, you find that decimal 1234 is equal to hexadecimal \$04D2.

Next, you divide that hexadecimal version of the number into two bytes: the most-significant byte (MSB) is \$04, and the least-significant byte (LSB) is \$D2. Divided that way, you end up with two 1-byte hexadecimal values: \$04 and \$D2.

Finally, convert those two sets of hexadecimal numbers into their decimal equivalents, treating them as two separate hexadecimal values. Thus \$04 converts to decimal 4, and \$D2 converts to 210.

The 2-byte version of decimal 1234 is thus 4 and 210, with 4 being the MSB and 210 being the LSB.

That takes care of the conversion of an ordinary decimal number into a 2-byte version, also in decimal. Now you must POKE these numbers into decimal addresses 16787 and 16788.

If you place the LSB of the 2-byte number into the lower-numbered address, the BASIC operation for satisfying the requirements of the example looks like this:

POKE 16787, 210 : POKE 16788, 4

No, it isn't a simple procedure to convert an ordinary decimal number into a 2-byte decimal format, but it's the price that must be paid for working with a byte-oriented machine in a decimal-oriented BASIC language.

TWO-BYTE DECIMAL TO CONVENTIONAL DECIMAL FORMAT

Suppose that you are analyzing a machine-language routine that is presented in a decimal-oriented, BASIC format. Under that condition, a 2-byte address appears as a set of two decimal numbers; if you want to get that pair of numbers into a conventional decimal format, you have to play with them a bit.

Consider an instance where 223 turns up as the LSB in decimal, and 104 is the MSB. What address, or 2-byte decimal number, do they represent?

First, convert both sets of numbers into their hexadecimal counterparts; decimal 223 = \$DF, and decimal 104 = \$68. Since \$DF is the MSB and \$68 is the LSB, the overall hexadecimal representation of that 2-byte decimal format is \$DF68.

All that remains to be done is to convert that hexadecimal number into its full decimal counterpart:

$$\text{\$DF68} = 24567 + 2048 + 208 + 15 = 26849$$

That's it—the conventional representation of the original 2-byte decimal values. The combination of decimal numbers 223 and 104 actually points to decimal 26849.

BINARY-TO-DECIMAL CONVERSION

In practice, most binary-to-decimal conversions are carried out with 1-byte (or 8-bit) binary numbers, although there are occasions when it is necessary to do the conversion from 2-byte (16-bit) numbers.

Fig. A-2 shows the breakdown of an 8-bit binary number. The positions are labeled 0 through 7, with zero indicating the least-significant bit position. Each of those 8-bit locations contains either a 0 or a 1.

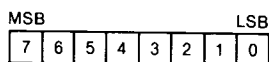


Fig. A-2. An 8-bit binary number.

Suppose that you want to POKE 01101011. But you have to use a decimal version of that binary number from BASIC. Here is how to go about determining that decimal version.

First, multiply the 0 or 1 in each bit location by 2^n , where n is the bit place in each case. Then simply add the results. (See the example in Fig. A-3.)

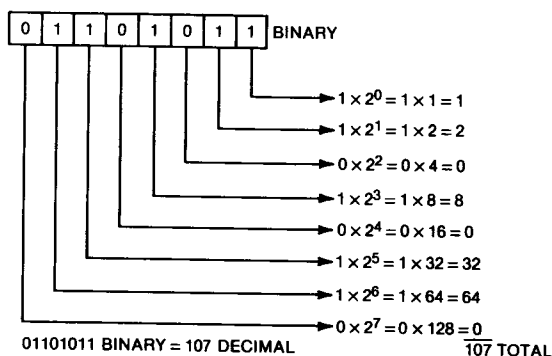


Fig. A-3. Converting binary values to decimal values.

The same idea applies to converting 16-bit binary to a decimal equivalent. The place values run from 0 to 15 in that case, and Table A-2 can help you determine those powers of 2.

BINARY-TO-HEXADECIMAL CONVERSION

Converting a binary number into a hexadecimal format is perhaps the simplest of all the conversion operations. All you have to do is group the binary number into sets of 4 bits each, beginning with the least-significant bit, and then find the hexadecimal value for each group. Table A-3 helps with the latter operation.

Table A-2. Powers of 2

n	2ⁿ
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768

Table A-3. Binary/Hexadecimal Conversion Table

Binary	Hex
0000	\$0
0001	\$1
0010	\$2
0011	\$3
0100	\$4
0101	\$5
0110	\$6
0111	\$7
1000	\$8
1001	\$9
1010	\$A
1011	\$B
1100	\$C
1101	\$D
1110	\$E
1111	\$F

Suppose the binary number is 10011101. There are two sets of 4 bits (or nibbles) here, 1001 and 1101. The hexadecimal equivalent is 9 for the first set, and D for the second set, as Table A-3 shows. Therefore, the hexadecimal version of that 8-bit binary number is \$9D.

The same procedure works equally well for 16-bit numbers; the only difference is that you end up with four hexadecimal characters instead of just two.

HEXADECIMAL-TO-BINARY CONVERSION

Converting a hexadecimal number to its binary form is a simple matter of applying Table A-3 to change each hexadecimal character into the appropriate groups of 4 binary bits.

Example: Convert address \$404D into a binary format. According to the table, that hexadecimal number can be represented as 0100 0000 0100 1101.

DECIMAL-TO-BINARY CONVERSION

There are several commonly cited algorithms for mathematically converting any decimal number into its binary format. But it is simpler in the long run, and probably more accurate, to use a two-step procedure.

The general idea is to convert the decimal number into its hexadecimal counterpart as described earlier in this appendix. Then convert the hexadecimal characters into their binary versions as described in the previous section.

Example: Convert 1234 decimal into binary. First, as described earlier, calculate the hexadecimal version of decimal 1234. Your answer should come out to be \$04D2. And that hexadecimal number, expressed in binary (from Table A-3) is 0000 0100 1101 0010. Thus 1234 is equal to binary 10011010010. You may include the five leading zeros if you wish.

A COMPLETE CONVERSION TABLE FOR DECIMAL 0-255

For readers who have no heart for doing a lot of number conversions, Table A-4 can come to the rescue. It is impractical to tabulate the conversions for the entire memory range of the ATARI system, but the range of values shown here will apply to any data byte.

**Table A-4. Decimal/Hexadecimal/Binary Table
for Decimal Values 0 Through 255**

DEC	HEX	BIN	DEC	HEX	BIN
0	\$00	00000000	32	\$20	00100000
1	\$01	00000001	33	\$21	00100001
2	\$02	00000010	34	\$22	00100010
3	\$03	00000011	35	\$23	00100011
4	\$04	00000100	36	\$24	00100100
5	\$05	00000101	37	\$25	00100101
6	\$06	00000110	38	\$26	00100110
7	\$07	00000111	39	\$27	00100111
8	\$08	00001000	40	\$28	00101000
9	\$09	00001001	41	\$29	00101001
10	\$0A	00001010	42	\$2A	00101010
11	\$0B	00001011	43	\$2B	00101011
12	\$0C	00001100	44	\$2C	00101100
13	\$0D	00001101	45	\$2D	00101101
14	\$0E	00001110	46	\$2E	00101110
15	\$0F	00001111	47	\$2F	00101111
16	\$10	00010000	48	\$30	00110000
17	\$11	00010001	49	\$31	00110001
18	\$12	00010010	50	\$32	00110010
19	\$13	00010011	51	\$33	00110011
20	\$14	00010100	52	\$34	00110100
21	\$15	00010101	53	\$35	00110101
22	\$16	00010110	54	\$36	00110110
23	\$17	00010111	55	\$37	00110111
24	\$18	00011000	56	\$38	00111000
25	\$19	00011001	57	\$39	00111001
26	\$1A	00011010	58	\$3A	00111010
27	\$1B	00011011	59	\$3B	00111011
28	\$1C	00011100	60	\$3C	00111100
29	\$1D	00011101	61	\$3D	00111101
30	\$1E	00011110	62	\$3E	00111110
31	\$1F	00011111	63	\$3F	00111111

**Table A-4—cont. Decimal/Hexadecimal/Binary Table
for Decimal Values 0 Through 255**

DEC	HEX	BIN	DEC	HEX	BIN
64	\$40	01000000	96	\$60	01100000
65	\$41	01000001	97	\$61	01100001
66	\$42	01000010	98	\$62	01100010
67	\$43	01000011	99	\$63	01100011
68	\$44	01000100	100	\$64	01100100
69	\$45	01000101	101	\$65	01100101
70	\$46	01000110	102	\$66	01100110
71	\$47	01000111	103	\$67	01100111
72	\$48	01001000	104	\$68	01101000
73	\$49	01001001	105	\$69	01101001
74	\$4A	01001010	106	\$6A	01101010
75	\$4B	01001011	107	\$6B	01101011
76	\$4C	01001100	108	\$6C	01101100
77	\$4D	01001101	109	\$6D	01101101
78	\$4E	01001110	110	\$6E	01101110
79	\$4F	01001111	111	\$6F	01101111
80	\$50	01010000	112	\$70	01110000
81	\$51	01010001	113	\$71	01110001
82	\$52	01010010	114	\$72	01110010
83	\$53	01010011	115	\$73	01110011
84	\$54	01010100	116	\$74	01110100
85	\$55	01010101	117	\$75	01110101
86	\$56	01010110	118	\$76	01110110
87	\$57	01010111	119	\$77	01110111
88	\$58	01011000	120	\$78	01111000
89	\$59	01011001	121	\$79	01111001
90	\$5A	01011010	122	\$7A	01111010
91	\$5B	01011011	123	\$7B	01111011
92	\$5C	01011100	124	\$7C	01111100
93	\$5D	01011101	125	\$7D	01111101
94	\$5E	01011110	126	\$7E	01111110
95	\$5F	01011111	127	\$7F	01111111

**Table A-4—cont. Decimal/Hexadecimal/Binary Table
for Decimal Values 0 Through 255**

DEC	HEX	BIN	DEC	HEX	BIN
128	\$80	10000000	160	\$A0	10100000
129	\$81	10000001	161	\$A1	10100001
130	\$82	10000010	162	\$A2	10100010
131	\$83	10000011	163	\$A3	10100011
132	\$84	10000100	164	\$A4	10100100
133	\$85	10000101	165	\$A5	10100101
134	\$86	10000110	166	\$A6	10100110
135	\$87	10000111	167	\$A7	10100111
136	\$88	10001000	168	\$A8	10101000
137	\$89	10001001	169	\$A9	10101001
138	\$8A	10001010	170	\$AA	10101010
139	\$8B	10001011	171	\$AB	10101011
140	\$8C	10001100	172	\$AC	10101100
141	\$8D	10001101	173	\$AD	10101101
142	\$8E	10001110	174	\$AE	10101110
143	\$8F	10001111	175	\$AF	10101111
144	\$90	10010000	176	\$B0	10110000
145	\$91	10010001	177	\$B1	10110001
146	\$92	10010010	178	\$B2	10110010
147	\$93	10010011	179	\$B3	10110011
148	\$94	10010100	180	\$B4	10110100
149	\$95	10010101	181	\$B5	10110101
150	\$96	10010110	182	\$B6	10110110
151	\$97	10010111	183	\$B7	10110111
152	\$98	10011000	184	\$B8	10111000
153	\$99	10011001	185	\$B9	10111001
154	\$9A	10011010	186	\$BA	10111010
155	\$9B	10011011	187	\$BB	10111011
156	\$9C	10011100	188	\$BC	10111100
157	\$9D	10011101	189	\$BD	10111101
158	\$9E	10011110	190	\$BE	10111110
159	\$9F	10011111	191	\$BF	10111111

**Table A-4—cont. Decimal/Hexadecimal/Binary Table
for Decimal Values 0 Through 255**

DEC	HEX	BIN	DEC	HEX	BIN
192	\$C0	11000000	224	\$E0	11100000
193	\$C1	11000001	225	\$E1	11100001
194	\$C2	11000010	226	\$E2	11100010
195	\$C3	11000011	227	\$E3	11100011
196	\$C4	11000100	228	\$E4	11100100
197	\$C5	11000101	229	\$E5	11100101
198	\$C6	11000110	230	\$E6	11100110
199	\$C7	11000111	231	\$E7	11100111
200	\$C8	11001000	232	\$E8	11101000
201	\$C9	11001001	233	\$E9	11101001
202	\$CA	11001010	234	\$EA	11101010
203	\$CB	11001011	235	\$EB	11101011
204	\$CC	11001100	236	\$EC	11101100
205	\$CD	11001101	237	\$ED	11101101
206	\$CE	11001110	238	\$EE	11101110
207	\$CF	11001111	239	\$EF	11101111
208	\$D0	11010000	240	\$F0	11110000
209	\$D1	11010001	241	\$F1	11110001
210	\$D2	11010010	242	\$F2	11110010
211	\$D3	11010011	243	\$F3	11110011
212	\$D4	11010100	244	\$F4	11110100
213	\$D5	11010101	245	\$F5	11110101
214	\$D6	11010110	246	\$F6	11110110
215	\$D7	11010111	247	\$F7	11110111
216	\$D8	11011000	248	\$F8	11111000
217	\$D9	11011001	249	\$F9	11111001
218	\$DA	11011010	250	\$FA	11111010
219	\$DB	11011011	251	\$FB	11111011
220	\$DC	11011100	252	\$FC	11111100
221	\$DD	11011101	253	\$FD	11111101
222	\$DE	11011110	254	\$FE	11111110
223	\$DF	11011111	255	\$FF	11111111

Appendix B

ATARI BASIC Reserved Words and Tokens

Chart B-1 lists the reserved words for ATARI BASIC. These must not be used as variable names.

Table B-1 shows the ATARI BASIC commands, operators, and functions that are tokenized in RAM as they are entered from the keyboard. BASIC programs can be saved on cassette tape or disk in this abbreviated format, or they can be saved as longer ATASCII character codes. See Chapter 7 for a discussion of the tokenizing procedure.

Chart B-1. ATARI BASIC Reserved Words

ATARI BASIC Reserved Words		
ABS	LEN	SGN
ADR	LET	SIN
AND	LIST	SOUND
ASC	LOAD	SQR
ATN	LOCATE	STATUS
BYE	LOG	STEP
CLOAD	LPRINT	STICK
CHR\$	NEW	STRIG
CLOG	NEXT	STOP
CLOSE	NOT	STR\$
CLR	NOTE	THEN
COLOR	ON	TO
COM	OPEN	TRAP
CONT	OR	USR
COS	PADDLE	VAL
CSAVE	PEEK	XIO
DATA	PLOT	
DEG	POINT	
DIM	POKE	
DOS	POP	
DRAWTO	POSITION	
END	PRINT	
ENTER	PTRIG	
EXP	PUT	
FOR	RAD	
FRE	READ	
GET	REM	
GOSUB	RESTORE	
GOTO	RETURN	
GRAPHICS	RND	
IF	RUN	
INPUT	SAVE	
INT	SETCOLOR	

**Table B-1. ATARI BASIC Command Keywords
and Their Decimal/Hexadecimal Tokens**

Command Tokens			Command Tokens		
Dec	Hex	Keyword	Dec	Hex	Keyword
0	\$00	REM	28	\$1C	POINT
1	\$01	DATA	29	\$1D	XIO
2	\$02	INPUT	30	\$1E	ON
3	\$03	COLOR	31	\$1F	POKE
4	\$04	LIST	32	\$20	PRINT
5	\$05	ENTER	33	\$21	RAD
6	\$06	LET	34	\$22	READ
7	\$07	IF	35	\$23	RESTORE
8	\$08	FOR	36	\$24	RETURN
9	\$09	NEXT	37	\$25	RUN
10	\$0A	GOTO	38	\$26	STOP
11	\$0B	GO TO	39	\$27	POP
12	\$0C	GOSUB	40	\$28	?
13	\$0D	TRAP	41	\$29	GET
14	\$0E	BYE	42	\$2A	PUT
15	\$0F	CONT	43	\$2B	GRAPHICS
16	\$10	COM	44	\$2C	PLOT
17	\$11	CLOSE	45	\$2D	POSITION
18	\$12	CLR	46	\$2E	DOS
19	\$13	DEG	47	\$2F	DRAWTO
20	\$14	DIM	48	\$30	SETCOLOR
21	\$15	END	49	\$31	LOCATE
22	\$16	NEW	50	\$32	SOUND
23	\$17	OPEN	51	\$33	LPRINT
24	\$18	LOAD	52	\$34	CSAVE
25	\$19	SAVE	53	\$35	CLOAD
26	\$1A	STATUS	54	\$36	implied LET
27	\$1B	NOTE	55	\$37	ERROR - (syntax)

**Table B-1—cont. ATARI BASIC Operator Keywords
and Their Decimal/Hexadecimal Tokens**

Operator Tokens			Operator Tokens		
Dec	Hex	Keyword	Dec	Hex	Keyword
14	\$E	num constant	37	\$25	+
15	\$F	string constant	38	\$26	-
16	\$10	not used	39	\$27	/
17	\$11	not used	40	\$28	NOT
18	\$12	,	41	\$29	OR
19	\$13	\$	42	\$2A	AND
20	\$14	: (end of statement)	43	\$2B	(
21	\$15	;	44	\$2C)
22	\$16	end of line	45	\$2D	= (num assign)
23	\$17	GOTO	46	\$2E	= (str assign)
24	\$18	GOSUB	47	\$2F	<= (string)
25	\$19	TO	48	\$30	<>
26	\$1A	STEP	49	\$31	>=
27	\$1B	THEN	50	\$32	<
28	\$1C	#	51	\$33	>
29	\$1D	<= (numeric)	52	\$34	=
30	\$1E	<>	53	\$35	+ (unary)
31	\$1F	>=	54	\$36	-
32	\$20	<	55	\$37	((as in a string)
33	\$21	>	56	\$38	((array)
34	\$22	=	57	\$39	((array)
35	\$23		58	\$3A	((function)
36	\$24	*	59	\$3B	((dimension)
			60	\$3C	, (as in an array)

**Table B-1—cont. ATARI BASIC Function Keywords
and Their Decimal/Hexadecimal Tokens**

Function Tokens			Function Tokens		
Dec	Hex	Keyword	Dec	Hex	Keyword
61	\$3D	STR\$	73	\$49	FRE
62	\$3E	CHR\$	74	\$4A	EXP
63	\$3F	USR	75	\$4B	LOG
64	\$40	ASC	76	\$4C	CLOG
65	\$41	VAL	77	\$4D	SQR
66	\$42	LEN	78	\$4E	SGN
67	\$43	ADR	79	\$4F	ABS
68	\$44	ATN	80	\$50	INT
69	\$45	COS	81	\$51	PADDL
70	\$46	PEEK	82	\$52	STICK
71	\$47	SIN	83	\$53	PTRIG
72	\$48	RND	84	\$54	STRIG

Appendix C

ATARI Character Codes

Table C-1 lists the decimal and hexadecimal codes for the ATARI ATASCII characters and control functions. The table also indicates the conventional ASCII characters and control operations and the ATARI keystrokes that generate the ATASCII codes.

Keystroke designations that are separated by a hyphen indicate simultaneous key depressions. For example, CTRL-2 means strike the 2 key while holding down the CTRL key.

Keystroke designations that are separated by a slash indicate a sequence of keystrokes. For example, ESC/BACK S means strike the ESC key followed by the BACK S key.

Table C-2 portrays the ATARI system's internal character set. It shows the decimal or hexadecimal value, the ATARI character and the starting and ending addresses of the ROM location for that character. Inverse characters are generated by setting the most-significant bit of the code to 1, or adding 128 to the decimal code.

Table C-1. The ATARI ATASCII Character Set



















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
0	\$00		NUL	CTRL-,
1	\$01		SOH	CTRL-A
2	\$02		STX	CTRL-B
3	\$03		ETX	CTRL-C
4	\$04		EOT	CTRL-D
5	\$05		ENQ	CTRL-E
6	\$06		ACK	CTRL-F
7	\$07		BEL	CTRL-G
8	\$08		BS	CTRL-H
9	\$09		HT	CTRL-I
10	\$0A		LF	CTRL-J
11	\$0B		VT	CTRL-K
12	\$0C		FF	CTRL-L
13	\$0D		CR	CTRL-M
14	\$0E		SO	CTRL-N
15	\$0F		SI	CTRL-O
16	\$10		DLE	CTRL-P
17	\$11		DC1	CTRL-Q

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
18	\$12		DC2	CTRL-R
19	\$13		DC3	CTRL-S
20	\$14		DC4	CTRL-T
21	\$15		NAK	CTRL-U
22	\$16		SYN	CTRL-V
23	\$17		ETB	CTRL-W
24	\$18		CAN	CTRL-X
25	\$19		EM	CTRL-Y
26	\$1A		SUB	CTRL-Z
27	\$1B		ESC	ESC/ESC
28	\$1C		FS	ESC/CTRL--
29	\$1D		GS	ESC/CTRL==
30	\$1E		RS	ESC/CTRL-+
31	\$1F		US	ESC/CTRL-*
32	\$20		(space)	Space Bar
33	\$21		!	SHIFT-1
34	\$22		"	SHIFT-2

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
35	\$23		#	SHIFT-3
36	\$24		\$	SHIFT-4
37	\$25		%	SHIFT-5
38	\$26		&	SHIFT-6
39	\$27		'	SHIFT-7
40	\$28		(SHIFT-9
41	\$29	)	SHIFT-0
42	\$2A		*	*
43	\$2B		+	+
44	\$2C		,	,
45	\$2D		-	-
46	\$2E		.	.
47	\$2F		/	/
48	\$30		0	0
49	\$31		1	1
50	\$32		2	2
51	\$33		3	3

Table C-1—cont. The ATARI ATASCII Character Set

Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
52	\$34	4	4	4
53	\$35	5	5	5
54	\$36	6	6	6
55	\$37	7	7	7
56	\$38	8	8	8
57	\$39	9	9	9
58	\$3A	:	:	SHIFT-;
59	\$3B	;	;	;
60	\$3C	<	<	<
61	\$3D	=	=	=
62	\$3E	>	>	>
63	\$3F	?	?	SHIFT-/
64	\$40	@	@	SHIFT-8
65	\$41	A	A	A
66	\$42	B	B	B
67	\$43	C	C	C
68	\$44	D	D	D

Table C-1—cont. The ATARI ATASCII Character Set

Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
69	\$45	E	E	E
70	\$46	F	F	F
71	\$47	G	G	G
72	\$48	H	H	H
73	\$49	I	I	I
74	\$4A	J	J	J
75	\$4B	K	K	K
76	\$4C	L	L	L
77	\$4D	M	M	M
78	\$4E	N	N	N
79	\$4F	O	O	O
80	\$50	P	P	P
81	\$51	Q	Q	Q
82	\$52	R	R	R
83	\$53	S	S	S
84	\$54	T	T	T
85	\$55	U	U	U

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
86	\$56		V	V
87	\$57		W	W
88	\$58		X	X
89	\$59		Y	Y
90	\$5A		Z	Z
91	\$5B		[SHIFT-;
92	\$5C		\	SHIFT-,
93	\$5D	]	SHIFT-+
94	\$5E		^	SHIFT-*
95	\$5F		_	SHIFT--
96	\$60		.	CTRL-.
97	\$61		a	LOWR A
98	\$62		b	LOWR B
99	\$63		c	LOWR C
100	\$64		d	LOWR D
101	\$65		e	LOWR E
102	\$66		f	LOWR F

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
103	\$67		g	LOWR G
104	\$68		h	LOWR H
105	\$69		i	LOWR I
106	\$6A		j	LOWR J
107	\$6B		k	LOWR K
108	\$6C		l	LOWR L
109	\$6D		m	LOWR M
110	\$6E		n	LOWR N
111	\$6F		o	LOWR O
112	\$70		p	LOWR P
113	\$71		q	LOWR Q
114	\$72		r	LOWR R
115	\$73		s	LOWR S
116	\$74		t	LOWR T
117	\$75		u	LOWR U
118	\$76		v	LOWR V
119	\$77		w	LOWR W

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
120	\$78		x	LOWR X
121	\$79		y	LOWR Y
122	\$7A		z	LOWR Z
123	\$7B		{	CTRL-;
124	\$7C			SHIFT-=
125	\$7D		}	ESC/CTRL-<
126	\$7E		~	ESC/BACK S
127	\$7F		DEL	ESC/TAB
128	\$80		NUL	CTRL-,
129	\$81		SOH	CTRL-A
130	\$82		STX	CTRL-B
131	\$83		ETX	CTRL-C
132	\$84		EOT	CTRL-D
133	\$85		ENQ	CTRL-E
134	\$86		ACK	CTRL-F
135	\$87		BEL	CTRL-G
136	\$88		BS	CTRL-H

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
137	\$89		HT	CTRL-I
138	\$8A		LF	CTRL-J
139	\$8B		VT	CTRL-K
140	\$8C		FF	CTRL-L
141	\$8D		CR	CTRL-M
142	\$8E		SO	CTRL-N
143	\$8F		SI	CTRL-O
144	\$90		DLE	CTRL-P
145	\$91		DC1	CTRL-Q
146	\$92		DC2	CTRL-R
147	\$93		DC3	CTRL-S
148	\$94		DC4	CTRL-T
149	\$95		NAK	CTRL-U
150	\$96		SYN	CTRL-V
151	\$97		ETB	CTRL-W
152	\$98		CAN	CTRL-X
153	\$99		EM	CTRL-Y

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
154	\$9A		SUB	CTRL-Z
155	\$9B		ESC	RETURN
156	\$9C		FS	ESC/SHIFT-BACK S
157	\$9D		GS	ESC/SHIFT->
158	\$9E		RS	ESC/CTRL-TAB
159	\$9F		US	ESC/SHIFT-TAB
160	\$A0		(space)	Space Bar
161	\$A1		!	SHIFT-1
162	\$A2		"	SHIFT-2
163	\$A3		#	SHIFT-3
164	\$A4		\$	SHIFT-4
165	\$A5		%	SHIFT-5
166	\$A6		&	SHIFT-6
167	\$A7		'	SHIFT-7
168	\$A8		(SHIFT-9
169	\$A9	)	SHIFT-0
170	\$AA		*	*

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
171	\$AB		+	+
172	\$AC		,	,
173	\$AD		-	-
174	\$AE		.	.
175	\$AF		/	/
176	\$B0		0	0
177	\$B1		1	1
178	\$B2		2	2
179	\$B3		3	3
180	\$B4		4	4
181	\$B5		5	5
182	\$B6		6	6
183	\$B7		7	7
184	\$B8		8	8
185	\$B9		9	9
186	\$BA		:	SHIFT-;
187	\$BB		'	'

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
188	\$BC		<	<
189	\$BD		=	=
190	\$BE		>	>
191	\$BF		?	SHIFT-/
192	\$C0		@	SHIFT-8
193	\$C1		A	A
194	\$C2		B	B
195	\$C3		C	C
196	\$C4		D	D
197	\$C5		E	E
198	\$C6		F	F
199	\$C7		G	G
200	\$C8		H	H
201	\$C9		I	I
202	\$CA		J	J
203	\$CB		K	K
204	\$CC		L	L

Table C-1—cont. The ATARI ATASCII Character Set


















Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
205	\$CD		M	M
206	\$CE		N	N
207	\$CF		O	O
208	\$D0		P	P
209	\$D1		Q	Q
210	\$D2		R	R
211	\$D3		S	S
212	\$D4		T	T
213	\$D5		U	U
214	\$D6		V	V
215	\$D7		W	W
216	\$D8		X	X
217	\$D9		Y	Y
218	\$DA		Z	Z
219	\$DB		[SHIFT-,
220	\$DC		\	SHIFT-+
221	\$DD	]	SHIFT-.

Table C-1—cont. The ATARI ATASCII Character Set

Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
222	\$DE		^	SHIFT-*
223	\$DF		—	SHIFT--
224	\$E0		\	CTRL-.
225	\$E1		a	LOWR A
226	\$E2		b	LOWR B
227	\$E3		c	LOWR C
228	\$E4		d	LOWR D
229	\$E5		e	LOWR E
230	\$E6		f	LOWR F
231	\$E7		g	LOWR G
232	\$E8		h	LOWR H
233	\$E9		i	LOWR I
234	\$EA		j	LOWR J
235	\$EB		k	LOWR K
236	\$EC		l	LOWR L
237	\$ED		m	LOWR M
238	\$EE		n	LOWR N

Table C-1—cont. The ATARI ATASCII Character Set


































Code		ATASCII Character	ASCII Character or Control	Keystroke(s)
Dec	Hex			
239	\$EF		o	LOWR O
240	\$F0		p	LOWR P
241	\$F1		q	LOWR Q
242	\$F2		r	LOWR R
243	\$F3		s	LOWR S
244	\$F4		t	LOWR T
245	\$F5		u	LOWR U
246	\$F6		v	LOWR V
247	\$F7		w	LOWR W
248	\$F8		x	LOWR X
249	\$F9		y	LOWR Y
250	\$FA		z	LOWR Z
251	\$FB		{	CTRL-;
252	\$FC			SHIFT-=
253	\$FD		\	ESC/CTRL-2
254	\$FE		~	ESC/CTRL-BACK S
255	\$FF		DEL	ESC/CTRL->

Table C-2. The ATARI System's Internal Character Set

Code		Character	ROM Map Addresses			
			Decimal		Hexadecimal	
Dec	Hex		Start	End	Start	End
0	\$00		57344	57351	\$E000	\$E007
1	\$01		57352	57359	\$E008	\$E00F
2	\$02		57360	57367	\$E010	\$E017
3	\$03		57368	57375	\$E018	\$E01F
4	\$04		57376	57383	\$E020	\$E027
5	\$05		57384	57391	\$E028	\$E02F
6	\$06		57392	57399	\$E030	\$E037
7	\$07		57400	57407	\$E038	\$E03F
8	\$08		57408	57415	\$E040	\$E047
9	\$09		57416	57423	\$E048	\$E04F
10	\$0A		57424	57431	\$E050	\$E057
11	\$0B		57432	57439	\$E058	\$E05F
12	\$0C		57440	57447	\$E060	\$E067
13	\$0D		57448	57455	\$E068	\$E06F
14	\$0E		57456	57463	\$E070	\$E077
15	\$0F		57464	57471	\$E078	\$E07F

**Table C-2—cont. The ATARI System's
Internal Character Set**

Code		Character	ROM Map Addresses			
			Decimal		Hexadecimal	
Dec	Hex		Start	End	Start	End
16	\$10	0	57472	57479	\$E080	\$E087
17	\$11	1	57480	57487	\$E088	\$E08F
18	\$12	2	57488	57495	\$E090	\$E097
19	\$13	3	57496	57503	\$E098	\$E09F
20	\$14	4	57504	57511	\$E0A0	\$E0A7
21	\$15	5	57512	57519	\$E0A8	\$E0AF
22	\$16	6	57520	57527	\$E0B0	\$E0B7
23	\$17	7	57528	57535	\$E0B8	\$E0BF
24	\$18	8	57536	57543	\$E0C0	\$E0C7
25	\$19	9	57544	57551	\$E0C8	\$E0CF
26	\$1A	÷	57552	57559	\$E0D0	\$E0D7
27	\$1B	×	57560	57567	\$E0D8	\$E0DF
28	\$1C	<	57568	57575	\$E0E0	\$E0E7
29	\$1D	=	57576	57583	\$E0E8	\$E0EF
30	\$1E	>	57584	57591	\$E0F0	\$E0F7
31	\$1F	?	57592	57599	\$E0F8	\$E0FF

















**Table C-2—cont. The ATARI System's
Internal Character Set**

Code		Character	ROM Map Addresses			
			Decimal		Hexadecimal	
Dec	Hex		Start	End	Start	End
32	\$20	@	57600	57607	\$E100	\$E107
33	\$21	A	57608	57615	\$E108	\$E10F
34	\$22	B	57616	57623	\$E110	\$E117
35	\$23	C	57624	57631	\$E118	\$E11F
36	\$24	D	57632	57639	\$E120	\$E127
37	\$25	E	57640	57647	\$E128	\$E12F
38	\$26	F	57648	57655	\$E130	\$E137
39	\$27	G	57656	57663	\$E138	\$E13F
40	\$28	H	57664	57671	\$E140	\$E147
41	\$29	I	57672	57679	\$E148	\$E14F
42	\$2A	J	57680	57687	\$E150	\$E157
43	\$2B	K	57688	57695	\$E158	\$E15F
44	\$2C	L	57696	57703	\$E160	\$E167
45	\$2D	M	57704	57711	\$E168	\$E16F
46	\$2E	N	57712	57719	\$E170	\$E177
47	\$2F	O	57720	57727	\$E178	\$E17F

















**Table C-2—cont. The ATARI System's
Internal Character Set**

Code		Character	ROM Map Addresses			
			Decimal		Hexadecimal	
Dec	Hex		Start	End	Start	End
48	\$30	P	57728	57735	\$E180	\$E187
49	\$31	Q	57736	57743	\$E188	\$E18F
50	\$32	R	57744	57751	\$E190	\$E197
51	\$33	S	57752	57759	\$E198	\$E19F
52	\$34	T	57760	57767	\$E1A0	\$E1A7
53	\$35	U	57768	57775	\$E1A8	\$E1AF
54	\$36	V	57776	57783	\$E1B0	\$E1B7
55	\$37	W	57784	57791	\$E1B8	\$E1BF
56	\$38	X	57792	57799	\$E1C0	\$E1C7
57	\$39	Y	57800	57807	\$E1C8	\$E1CF
58	\$3A	Z	57808	57815	\$E1D0	\$E1D7
59	\$3B	[57816	57823	\$E1D8	\$E1DF
60	\$3C	↘	57824	57831	\$E1E0	\$E1E7
61	\$3D]	57832	57839	\$E1E8	\$E1EF
62	\$3E	↗	57840	57847	\$E1F0	\$E1F7
63	\$3F	...	57848	57855	\$E1F8	\$E1FF

















**Table C-2—cont. The ATARI System's
Internal Character Set**

Code		Character	ROM Map Addresses			
			Decimal		Hexadecimal	
Dec	Hex		Start	End	Start	End
64	\$40		57856	57863	\$E200	\$E207
65	\$41		57864	57871	\$E208	\$E20F
66	\$42		57872	57879	\$E210	\$E217
67	\$43		57880	57887	\$E218	\$E21F
68	\$44		57888	57895	\$E220	\$E227
69	\$45		57896	57903	\$E228	\$E22F
70	\$46		57904	57911	\$E230	\$E237
71	\$47		57912	57919	\$E238	\$E23F
72	\$48		57920	57927	\$E240	\$E247
73	\$49		57928	57935	\$E248	\$E24F
74	\$4A		57936	57943	\$E250	\$E257
75	\$4B		57944	57951	\$E258	\$E25F
76	\$4C		57952	57959	\$E260	\$E267
77	\$4D		57960	57967	\$E268	\$E26F
78	\$4E		57968	57975	\$E270	\$E277
79	\$4F		57976	57983	\$E278	\$E27F

















**Table C-2—cont. The ATARI System's
Internal Character Set**

Code		Character	ROM Map Addresses			
			Decimal		Hexadecimal	
Dec	Hex		Start	End	Start	End
80	\$50		57984	57991	\$E280	\$E287
81	\$51		57992	57999	\$E288	\$E28F
82	\$52		58000	58007	\$E290	\$E297
83	\$53		58008	58015	\$E298	\$E29F
84	\$54		58016	58023	\$E2A0	\$E2A7
85	\$55		58024	58031	\$E2A8	\$E2AF
86	\$56		58032	58039	\$E2B0	\$E2B7
87	\$57		58040	58047	\$E2B8	\$E2BF
88	\$58		58048	58055	\$E2C0	\$E2C7
89	\$59		58056	58063	\$E2C8	\$E2CF
90	\$5A		58064	58071	\$E2D0	\$E2D7
91	\$5B		58072	58079	\$E2D8	\$E2DF
92	\$5C		58080	58087	\$E2E0	\$E2E7
93	\$5D		58088	58095	\$E2E8	\$E2EF
94	\$5E		58096	58103	\$E2F0	\$E2F7
95	\$5F		58104	58111	\$E2F8	\$E2FF

**Table C-2—cont. The ATARI System's
Internal Character Set**

Code		Character	ROM Map Addresses			
			Decimal		Hexadecimal	
Dec	Hex		Start	End	Start	End
96	\$60		58112	58119	\$E300	\$E307
97	\$61		58120	58127	\$E308	\$E30F
98	\$62		58128	58135	\$E310	\$E317
99	\$63		58136	58143	\$E318	\$E31F
100	\$64		58144	58151	\$E320	\$E327
101	\$65		58152	58159	\$E328	\$E32F
102	\$66		58160	58167	\$E330	\$E337
103	\$67		58168	58175	\$E338	\$E33F
104	\$68		58176	58183	\$E340	\$E347
105	\$69		58184	58191	\$E348	\$E34F
106	\$6A		58192	58199	\$E350	\$E357
107	\$6B		58200	58207	\$E358	\$E35F
108	\$6C		58208	58215	\$E360	\$E367
109	\$6D		58216	58223	\$E368	\$E36F
110	\$6E		58224	58231	\$E370	\$E377
111	\$6F		58232	58239	\$E378	\$E37F

**Table C-2—cont. The ATARI System's
Internal Character Set**

Code		Character	ROM Map Addresses			
			Decimal		Hexadecimal	
Dec	Hex		Start	End	Start	End
112	\$70		58240	58247	\$E380	\$E387
113	\$71		58248	58255	\$E388	\$E38F
114	\$72		58256	58263	\$E390	\$E397
115	\$73		58264	58271	\$E398	\$E39F
116	\$74		58272	58279	\$E3A0	\$E3A7
117	\$75		58280	58287	\$E3A8	\$E3AF
118	\$76		58288	58295	\$E3B0	\$E3B7
119	\$77		58296	58303	\$E3B0	\$E3BF
120	\$78		58304	58311	\$E3C0	\$E3C7
121	\$79		58312	58319	\$E3C8	\$E3CF
122	\$7A		58320	58327	\$E3D0	\$E3D7
123	\$7B		58328	58335	\$E3D8	\$E3DF
124	\$7C		58336	58343	\$E3E0	\$E3E7
125	\$7D		58344	58351	\$E3E8	\$E3EF
126	\$7E		58352	58359	\$E3F0	\$E3F7
127	\$7F		59360	58367	\$E3F8	\$E3FF

Appendix D

ATARI Keyboard Codes

Every keystroke and appropriate combination of keystrokes produces a character code that represents that keyboard action. The tables shown here list those keystrokes and codes that are subsequently generated within the ATARI system.

Tables D-1 through D-3 list the information by showing the keystroke and corresponding decimal and hexadecimal codes. Table D-4 lists the same information, but it is organized according to the numerical sequence of the codes.

**Table D-1. Keystrokes and Corresponding Key Codes
(Single-Key Operations)**

Key	Code		Key	Code	
	Dec	Hex		Dec	Hex
Space Bar	32	\$20	9	57	\$39
!	33	\$21	:	58	\$3A
"	34	\$22	;	59	\$3B
#	35	\$23	<	60	\$3C
\$	36	\$24	=	61	\$3D
%	37	\$25	>	62	\$3E
&	38	\$26	?	63	\$3F
'	39	\$27	@	64	\$40
(40	\$28	A	65	\$41
)	41	\$29	B	66	\$42
*	42	\$2A	C	67	\$43
+	43	\$2B	D	68	\$44
,	44	\$2C	E	69	\$45
-	45	\$2D	F	70	\$46
.	46	\$2E	G	71	\$47
/	47	\$2F	H	72	\$48
0	48	\$30	I	73	\$49
1	49	\$31	J	74	\$4A
2	50	\$32	K	75	\$4B
3	51	\$33	L	76	\$4C
4	52	\$34	M	77	\$4D
5	53	\$35	N	78	\$4E
6	54	\$36	O	79	\$4F
7	55	\$37	P	80	\$50
8	56	\$38	Q	81	\$51

**Table D-1—cont. Keystrokes and Corresponding
Key Codes (Single-Key Operations)**

Key	Code		Key	Code	
	Dec	Hex		Dec	Hex
R	82	\$52	l	108	\$6C
S	83	\$53	m	109	\$6D
T	84	\$54	n	110	\$6E
U	85	\$55	o	111	\$6F
V	86	\$56	p	112	\$70
W	87	\$57	q	113	\$71
X	88	\$58	r	114	\$72
Y	89	\$59	s	115	\$73
Z	90	\$5A	t	116	\$74
[91	\$5B	u	117	\$75
/	92	\$5C	v	118	\$76
]	93	\$5D	w	119	\$77
^	94	\$5E	x	120	\$78
_	95	\$5F	y	121	\$79
a	97	\$61	z	122	\$7A
b	98	\$62		124	\$7C
c	99	\$63	ESC	127	\$1B
d	100	\$64	CLEAR	125	\$7D
e	101	\$65	BACK S	126	\$7E
f	102	\$66	TAB	127	\$7F
g	103	\$67	RETURN	155	\$9B
h	104	\$68	DELETE	156	\$9C
i	105	\$69	INSERT	157	\$9D
j	106	\$6A	TAB	158	\$9E
k	107	\$6B	SET	159	\$9F

**Table D-2. Keystrokes and Corresponding Key Codes
(CTRL-Key Operations)**

Key	Code		Key	Code	
	Dec	Hex		Dec	Hex
CTRL-,	0	\$00	CTRL-S	19	\$13
CTRL-A	1	\$01	CTRL-T	20	\$14
CTRL-B	2	\$02	CTRL-U	21	\$15
CTRL-C	3	\$03	CTRL-V	22	\$16
CTRL-D	4	\$04	CTRL-W	23	\$17
CTRL-E	5	\$05	CTRL-X	24	\$18
CTRL-F	6	\$06	CTRL-Y	25	\$19
CTRL-G	7	\$07	CTRL-Z	26	\$1A
CTRL-H	8	\$08	CTRL-	28	\$1C
CTRL-I	9	\$09	CTRL-	29	\$1D
CTRL-J	10	\$0A	CTRL-	30	\$1E
CTRL-K	11	\$0B	CTRL-	31	\$1F
CTRL-L	12	\$0C	CTRL-	96	\$60
CTRL-M	13	\$0D	CTRL-;	123	\$7B
CTRL-N	14	\$0E	CTRL-2	253	\$FD
CTRL-O	15	\$0F	CTRL-3	155	\$9B
CTRL-P	16	\$10	CTRL-DELETE	254	\$FE
CTRL-Q	17	\$11	CTRL-INSERT	255	\$FF
CTRL-R	18	\$12			

NOTE: CTRL-key operations that are not indicated here will generate the same codes as their non-CTRL counterparts.

**Table D-3. Keystrokes and Corresponding Key Codes
(Inverse-Key Operations)**

Key	Code		Key	Code	
	Dec	Hex		Dec	Hex
<inv.> CTRL-,	128	\$80	<inv.> 1	177	\$B1
<inv.> CTRL-A	129	\$81	<inv.> 2	178	\$B2
<inv.> CTRL-B	130	\$82	<inv.> 3	179	\$B3
<inv.> CTRL-C	131	\$83	<inv.> 4	180	\$B4
<inv.> CTRL-D	132	\$84	<inv.> 5	181	\$B5
<inv.> CTRL-E	133	\$85	<inv.> 6	182	\$B6
<inv.> CTRL-F	134	\$86	<inv.> 7	183	\$B7
<inv.> CTRL-G	135	\$87	<inv.> 8	184	\$B8
<inv.> CTRL-H	136	\$88	<inv.> 9	185	\$B9
<inv.> CTRL-I	137	\$89	<inv.> :	186	\$BA
<inv.> CTRL-J	138	\$8A	<inv.> ;	187	\$BB
<inv.> CTRL-K	139	\$8B	<inv.> <	188	\$BC
<inv.> CTRL-L	140	\$8C	<inv.> =	189	\$BD
<inv.> CTRL-M	141	\$8D	<inv.> >	190	\$BE
<inv.> CTRL-N	142	\$8E	<inv.> ?	191	\$BF
<inv.> CTRL-O	143	\$8F	<inv.> @	192	\$C0
<inv.> CTRL-P	144	\$90	<inv.> A	193	\$C1
<inv.> CTRL-Q	145	\$91	<inv.> B	194	\$C2
<inv.> CTRL-R	146	\$92	<inv.> C	195	\$C3
<inv.> CTRL-S	147	\$93	<inv.> D	196	\$C4
<inv.> CTRL-T	148	\$94	<inv.> E	197	\$C5
<inv.> CTRL-U	149	\$95	<inv.> F	198	\$C6
<inv.> CTRL-V	150	\$96	<inv.> G	199	\$C7
<inv.> CTRL-W	151	\$97	<inv.> H	200	\$C8
<inv.> CTRL-X	152	\$98	<inv.> I	201	\$C9
<inv.> CTRL-Y	153	\$99	<inv.> J	202	\$CA
<inv.> CTRL-Z	154	\$9A	<inv.> K	203	\$CB
<inv.> Spc Bar	160	\$A0	<inv.> L	204	\$CC
<inv.> !	161	\$A1	<inv.> M	205	\$CD
<inv.> "	162	\$A2	<inv.> N	206	\$CE
<inv.> #	163	\$A3	<inv.> O	207	\$CF
<inv.> \$	164	\$A4	<inv.> P	208	\$D0
<inv.> %	165	\$A5	<inv.> Q	209	\$D1
<inv.> &	166	\$A6	<inv.> R	210	\$D2
<inv.> '	167	\$A7	<inv.> S	211	\$D3
<inv.> (168	\$A8	<inv.> T	212	\$D4
<inv.>)	169	\$A9	<inv.> U	213	\$D5
<inv.> *	170	\$AA	<inv.> V	214	\$D6
<inv.> +	171	\$AB	<inv.> W	215	\$D7
<inv.> ,	172	\$AC	<inv.> X	216	\$D8
<inv.> -	173	\$AD	<inv.> Y	217	\$D9
<inv.> .	174	\$AE	<inv.> Z	218	\$DA
<inv.> /	175	\$AF	<inv.> [219	\$DB
<inv.> 0	176	\$B0	<inv.> \	220	\$DC

**Table D-3—cont. Keystrokes and Corresponding
Key Codes (Inverse-Key Operations)**

Key	Code		Key	Code	
	Dec	Hex		Dec	Hex
<inv.> j	221	\$DD	<inv.> n	238	\$EE
<inv.> ^	222	\$DE	<inv.> o	239	\$EF
<inv.> —	223	\$DF	<inv.> p	240	\$F0
<inv.> a	225	\$E1	<inv.> q	241	\$F1
<inv.> b	226	\$E2	<inv.> r	242	\$F2
<inv.> c	227	\$E3	<inv.> s	243	\$F3
<inv.> d	228	\$E4	<inv.> t	244	\$F4
<inv.> e	229	\$E5	<inv.> u	245	\$F5
<inv.> f	230	\$E6	<inv.> v	246	\$F6
<inv.> g	231	\$E7	<inv.> w	247	\$F7
<inv.> h	232	\$E8	<inv.> x	248	\$F8
<inv.> i	233	\$E9	<inv.> y	249	\$F9
<inv.> j	234	\$EA	<inv.> z	250	\$FA
<inv.> k	235	\$EB	<inv.>	252	\$FC
<inv.> l	236	\$EC	<inv.> CTRL-.	224	\$E0
<inv.> m	237	\$ED	<inv.> CTRL-;	251	\$FB

**Table D-4. Keystrokes and Corresponding Key Codes
Arranged With Key Codes in Numerical Order**

Code		Key	Code		Key
Dec	Hex		Dec	Hex	
0	\$00	CTRL-	43	\$2B	+
1	\$01	CTRL-A	44	\$2C	,
2	\$02	CTRL-B	45	\$2D	-
3	\$03	CTRL-C	46	\$2E	.
4	\$04	CTRL-D	47	\$2F	/
5	\$05	CTRL-E	48	\$30	0
6	\$06	CTRL-F	49	\$31	1
7	\$07	CTRL-G	50	\$32	2
8	\$08	CTRL-H	51	\$33	3
9	\$09	CTRL-I	52	\$34	4
10	\$0A	CTRL-J	53	\$35	5
11	\$0B	CTRL-K	54	\$36	6
12	\$0C	CTRL-L	55	\$37	7
13	\$0D	CTRL-M	56	\$38	8
14	\$0E	CTRL-N	57	\$39	9
15	\$0F	CTRL-O	58	\$3A	:
16	\$10	CTRL-P	59	\$3B	;
17	\$11	CTRL-Q	60	\$3C	<
18	\$12	CTRL-R	61	\$3D	=
19	\$13	CTRL-S	62	\$3E	>
20	\$14	CTRL-T	63	\$3F	?
21	\$15	CTRL-U	64	\$40	@
22	\$16	CTRL-V	65	\$41	A
23	\$17	CTRL-W	66	\$42	B
24	\$18	CTRL-X	67	\$43	C
25	\$19	CTRL-Y	68	\$44	D
26	\$1A	CTRL-Z	69	\$45	E
27	\$1B	ESC	70	\$46	F
28	\$1C	CTRL-	71	\$47	G
29	\$1D	CTRL-	72	\$48	H
30	\$1E	CTRL-	73	\$49	I
31	\$1F	CTRL-	74	\$4A	J
32	\$20	Space Bar	75	\$4B	K
33	\$21	!	76	\$4C	L
34	\$22	"	77	\$4D	M
35	\$23	#	78	\$4E	N
36	\$24	\$	79	\$4F	O
37	\$25	%	80	\$50	P
38	\$26	&	81	\$51	Q
39	\$27	'	82	\$52	R
40	\$28	(83	\$53	S
41	\$29)	84	\$54	T
42	\$2A	*	85	\$55	U

Table D-4—cont. Keystrokes and Corresponding Key Codes Arranged With Key Codes in Numerical Order

Code		Key	Code		Key
Dec	Hex		Dec	Hex	
86	\$56	V	129	\$81	<inv.> CTRL-A
87	\$57	W	130	\$82	<inv.> CTRL-B
88	\$58	X	131	\$83	<inv.> CTRL-C
89	\$59	Y	132	\$84	<inv.> CTRL-D
90	\$5A	Z	133	\$85	<inv.> CTRL-E
91	\$5B	[134	\$86	<inv.> CTRL-F
92	\$5C	\	135	\$87	<inv.> CTRL-G
93	\$5D]	136	\$88	<inv.> CTRL-H
94	\$5E	^	137	\$89	<inv.> CTRL-I
95	\$5F	_	138	\$8A	<inv.> CTRL-J
96	\$60	CTRL-.	139	\$8B	<inv.> CTRL-K
97	\$61	a	140	\$8C	<inv.> CTRL-L
98	\$62	b	141	\$8D	<inv.> CTRL-M
99	\$63	c	142	\$8E	<inv.> CTRL-N
100	\$64	d	143	\$8F	<inv.> CTRL-O
101	\$65	e	144	\$90	<inv.> CTRL-P
102	\$66	f	145	\$91	<inv.> CTRL-Q
103	\$67	g	146	\$92	<inv.> CTRL-R
104	\$68	h	147	\$93	<inv.> CTRL-S
105	\$69	i	148	\$94	<inv.> CTRL-T
106	\$6A	j	149	\$95	<inv.> CTRL-U
107	\$6B	k	150	\$96	<inv.> CTRL-V
108	\$6C	l	151	\$97	<inv.> CTRL-W
109	\$6D	m	152	\$98	<inv.> CTRL-X
110	\$6E	n	153	\$99	<inv.> CTRL-Y
111	\$6F	o	154	\$9A	<inv.> CTRL-Z
112	\$70	p	155	\$9B	RETURN and CTRL-3
113	\$71	q	156	\$9C	DELETE
114	\$72	r	157	\$9D	INSERT
115	\$73	s	158	\$9E	TAB
116	\$74	t	159	\$9F	SET
117	\$75	u	160	\$A0	<inv.> Space
118	\$76	v	161	\$A1	<inv.> !
119	\$77	w	162	\$A2	<inv.> "
120	\$78	x	163	\$A3	<inv.> #
121	\$79	y	164	\$A4	<inv.> \$
122	\$7A	z	165	\$A5	<inv.> %
123	\$7B	CTRL-;	166	\$A6	<inv.> &
124	\$7C		167	\$A7	<inv.> '
125	\$7D	CLEAR	168	\$A8	<inv.> (
126	\$7E	BACK S	169	\$A9	<inv.>)
127	\$7F	TAB	170	\$AA	<inv.> *
128	\$80	<inv.> CTRL-,	171	\$AB	<inv.> +

Table D-4—cont. Keystrokes and Corresponding Key Codes Arranged With Key Codes in Numerical Order

Code		Key	Code		Key
Dec	Hex		Dec	Hex	
172	\$AC	<inv.> ,	214	\$D6	<inv.> V
173	\$AD	<inv.> -	215	\$D7	<inv.> W
174	\$AE	<inv.> .	216	\$D8	<inv.> X
175	\$AF	<inv.> /	217	\$D9	<inv.> Y
176	\$B0	<inv.> 0	218	\$DA	<inv.> Z
177	\$B1	<inv.> 1	219	\$DB	<inv.> [
178	\$B2	<inv.> 2	220	\$DC	<inv.> \
179	\$B3	<inv.> 3	221	\$DD	<inv.>]
180	\$B4	<inv.> 4	222	\$DE	<inv.> ^
181	\$B5	<inv.> 5	223	\$DF	<inv.> _
182	\$B6	<inv.> 6	224	\$E0	<inv.> CTRL-
183	\$B7	<inv.> 7	225	\$E1	<inv.> a
184	\$B8	<inv.> 8	226	\$E2	<inv.> b
185	\$B9	<inv.> 9	227	\$E3	<inv.> c
186	\$BA	<inv.> :	228	\$E4	<inv.> d
187	\$BB	<inv.> ;	229	\$E5	<inv.> e
188	\$BC	<inv.> <	230	\$E6	<inv.> f
189	\$BD	<inv.> =	231	\$E7	<inv.> g
190	\$BE	<inv.> >	232	\$E8	<inv.> h
191	\$BF	<inv.> ?	233	\$E9	<inv.> i
192	\$C0	<inv.> @	234	\$EA	<inv.> j
193	\$C1	<inv.> A	235	\$EB	<inv.> k
194	\$C2	<inv.> B	236	\$EC	<inv.> l
195	\$C3	<inv.> C	237	\$ED	<inv.> m
196	\$C4	<inv.> D	238	\$EE	<inv.> n
197	\$C5	<inv.> E	239	\$EF	<inv.> o
198	\$C6	<inv.> F	240	\$F0	<inv.> p
199	\$C7	<inv.> G	241	\$F1	<inv.> q
200	\$C8	<inv.> H	242	\$F2	<inv.> r
201	\$C9	<inv.> I	243	\$F3	<inv.> s
202	\$CA	<inv.> J	244	\$F4	<inv.> t
203	\$CB	<inv.> K	245	\$F5	<inv.> u
204	\$CC	<inv.> L	246	\$F6	<inv.> v
205	\$CD	<inv.> M	247	\$F7	<inv.> w
206	\$CE	<inv.> N	248	\$F8	<inv.> x
207	\$CF	<inv.> O	249	\$F9	<inv.> y
208	\$D0	<inv.> P	250	\$FA	<inv.> z
209	\$D1	<inv.> Q	251	\$FB	<inv.> CTRL-;
210	\$D2	<inv.> R	252	\$FC	<inv.>
211	\$D3	<inv.> S	253	\$FD	CTRL-2
212	\$D4	<inv.> T	254	\$FE	CTRL-DELETE
213	\$D5	<inv.> U	255	\$FF	CTRL-INSERT

Appendix E

Screen RAM Addressing Ranges for the ATARI Screen Modes

The tables in this appendix cite the starting and ending addresses for the screen RAM. Modes included here are 0 through 8 and their full-screen counterparts, 17 through 24.

**Table E-1. Absolute Addresses for ATARI
BASIC Mode-0 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40000	40039	\$9C40	\$9C67
Row 1	40040	40079	\$9C68	\$9C8F
Row 2	40080	40119	\$9C90	\$9CB7
Row 3	40120	40159	\$9CB8	\$9CDF
Row 4	40160	40199	\$9CE0	\$9D07
Row 5	40200	40239	\$9D08	\$9D2F
Row 6	40240	40279	\$9D30	\$9D57
Row 7	40280	40319	\$9D58	\$9D7F
Row 8	40320	40359	\$9D80	\$9DA7
Row 9	40360	40399	\$9DA8	\$9DCF
Row 10	40400	40439	\$9DD0	\$9DF7
Row 11	40440	40479	\$9DF8	\$9E1F
Row 12	40480	40519	\$9E20	\$9E47
Row 13	40520	40559	\$9E48	\$9E6F
Row 14	40560	40599	\$9E70	\$9E97
Row 15	40600	40639	\$9E98	\$9EBF
Row 16	40640	40679	\$9EC0	\$9EE7
Row 17	40680	40719	\$9EE8	\$9F0F
Row 18	40720	40759	\$9F10	\$9F37
Row 19	40760	40799	\$9F38	\$9F5F
Row 20	40800	40839	\$9F60	\$9F87
Row 21	40840	40879	\$9F88	\$9FAF
Row 22	40880	40919	\$9FB0	\$9FD7
Row 23	40920	40959	\$9FD8	\$9FFF

**Table E-2. Absolute Addresses for ATARI
BASIC Mode-1 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40320	40339	\$9D80	\$9D93
Row 1	40340	40359	\$9D94	\$9DA7
Row 2	40360	40379	\$9DA8	\$9DBB
Row 3	40380	40399	\$9DBC	\$9DCF
Row 4	40400	40419	\$9DD0	\$9DE3
Row 5	40420	40439	\$9DE4	\$9DF7
Row 6	40440	40459	\$9DF8	\$9E0B
Row 7	40460	40479	\$9E0C	\$9E1F
Row 8	40480	40499	\$9E20	\$9E33
Row 9	40500	40519	\$9E34	\$9E47
Row 10	40520	40539	\$9E48	\$9E5B
Row 11	40540	40559	\$9E5C	\$9E6F
Row 12	40560	40579	\$9E70	\$9E83
Row 13	40580	40599	\$9E84	\$9E97
Row 14	40600	40619	\$9E98	\$9EAB
Row 15	40620	40639	\$9EAC	\$9EBF
Row 16	40640	40659	\$9EC0	\$9ED3
Row 17	40660	40679	\$9ED4	\$9EE7
Row 18	40680	40699	\$9EE8	\$9EFB
Row 19	40700	40719	\$9EFC	\$9F0F
Text Window Begins Here				
Row 0	40800	40839	\$9F60	\$9F87
Row 1	40840	40879	\$9F88	\$9FAF
Row 2	40880	40919	\$9FB0	\$9FD7
Row 3	40920	40959	\$9FD8	\$9FFF

**Table E-3. Absolute Addresses for ATARI
BASIC Mode-17 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40320	40339	\$9D80	\$9D93
Row 1	40340	40359	\$9D94	\$9DA7
Row 2	40360	40379	\$9DA8	\$9DBB
Row 3	40380	40399	\$9DBC	\$9DCF
Row 4	40400	40419	\$9DD0	\$9DE3
Row 5	40420	40439	\$9DE4	\$9DF7
Row 6	40440	40459	\$9DF8	\$9E0B
Row 7	40460	40479	\$9E0C	\$9E1F
Row 8	40480	40499	\$9E20	\$9E33
Row 9	40500	40519	\$9E34	\$9E47
Row 10	40520	40539	\$9E48	\$9E5B
Row 11	40540	40559	\$9E5C	\$9E6F
Row 12	40560	40579	\$9E70	\$9E83
Row 13	40580	40599	\$9E84	\$9E97
Row 14	40600	40619	\$9E98	\$9EAB
Row 15	40620	40639	\$9EAC	\$9EBF
Row 16	40640	40659	\$9EC0	\$9ED3
Row 17	40660	40679	\$9ED4	\$9EE7
Row 18	40680	40699	\$9EE8	\$9EFB
Row 19	40700	40719	\$9EFC	\$9F0F
Row 20	40720	40739	\$9F10	\$9F23
Row 21	40740	40759	\$9F24	\$9F37
Row 22	40760	40779	\$9F38	\$9F4B
Row 23	40780	40799	\$9F4C	\$9F5F

**Table E-4. Absolute Addresses for ATARI
BASIC Mode-2 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40560	40579	\$9E70	\$9E83
Row 1	40580	40599	\$9E84	\$9E97
Row 2	40600	40619	\$9E98	\$9EAB
Row 3	40620	40639	\$9EAC	\$9EBF
Row 4	40640	40659	\$9EC0	\$9ED3
Row 5	40660	40679	\$9ED4	\$9EE7
Row 6	40680	40699	\$9EE8	\$9EFB
Row 7	40700	40719	\$9EFC	\$9F0F
Row 8	40720	40739	\$9F10	\$9F23
Row 9	40740	40759	\$9F24	\$9F37
Text Window Addressing Begins Here				
Row 0	40800	40839	\$9F60	\$9F87
Row 1	40840	40879	\$9F88	\$9FAF
Row 2	40880	40919	\$9FB0	\$9FD7
Row 3	40920	40959	\$9FD8	\$9FFF

**Table E-5. Absolute Addresses for ATARI
BASIC Mode-18 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40560	40579	\$9E70	\$9E83
Row 1	40580	40599	\$9E84	\$9E97
Row 2	40600	40619	\$9E98	\$9EAB
Row 3	40620	40639	\$9EAC	\$9EBF
Row 4	40640	40659	\$9EC0	\$9ED3
Row 5	40660	40679	\$9ED4	\$9EE7
Row 6	40680	40699	\$9EE8	\$9EFB
Row 7	40700	40719	\$9EFC	\$9F0F
Row 8	40720	40739	\$9F10	\$9F23
Row 9	40740	40759	\$9F24	\$9F37
Row 10	40760	40779	\$9F38	\$9F4B
Row 11	40780	40799	\$9F4C	\$9F5F

**Table E-6. Absolute Addresses for ATARI
BASIC Mode-3 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40560	40569	\$9E70	\$9E79
Row 1	40570	40579	\$9E7A	\$9E83
Row 2	40580	40589	\$9E84	\$9E8D
Row 3	40590	40599	\$9E8E	\$9E97
Row 4	40600	40609	\$9E98	\$9EA1
Row 5	40610	40619	\$9EA2	\$9EAB
Row 6	40620	40629	\$9EAC	\$9EB5
Row 7	40630	40639	\$9EB6	\$9EBF
Row 8	40640	40649	\$9EC0	\$9EC9
Row 9	40650	40659	\$9ECA	\$9ED3
Row 10	40660	40669	\$9ED4	\$9EDD
Row 11	40670	40679	\$9EDE	\$9EE7
Row 12	40680	40689	\$9EE8	\$9EF1
Row 13	40690	40699	\$9EF2	\$9EFB
Row 14	40700	40709	\$9EFC	\$9F05
Row 15	40710	40719	\$9F06	\$9F0F
Row 16	40720	40729	\$9F10	\$9F19
Row 17	40730	40739	\$9F1A	\$9F23
Row 18	40740	40749	\$9F24	\$9F2D
Row 19	40750	40759	\$9F2E	\$9F37
Text window addressing begins here				
Row 0	40800	40839	\$9F60	\$9F87
Row 1	40840	40879	\$9F88	\$9FAF
Row 2	40880	40919	\$9FB0	\$9FD7
Row 3	40920	40959	\$9FD8	\$9FFF

**Table E-7. Absolute Addresses for ATARI
BASIC Mode-19 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40560	40569	\$9E70	\$9E79
Row 1	40570	40579	\$9E7A	\$9E83
Row 2	40580	40589	\$9E84	\$9E8D
Row 3	40590	40599	\$9E8E	\$9E97
Row 4	40600	40609	\$9E98	\$9EA1
Row 5	40610	40619	\$9EA2	\$9EAB
Row 6	40620	40629	\$9EAC	\$9EB5
Row 7	40630	40639	\$9EB6	\$9EBF
Row 8	40640	40649	\$9ECO	\$9EC9
Row 9	40650	40659	\$9ECA	\$9ED3
Row 10	40660	40669	\$9ED4	\$9EDD
Row 11	40670	40679	\$9EDE	\$9EE7
Row 12	40680	40689	\$9EE8	\$9EFI
Row 13	40690	40699	\$9EF2	\$9EFB
Row 14	40700	40709	\$9EFC	\$9F05
Row 15	40710	40719	\$9F06	\$9F0F
Row 16	40720	40729	\$9F10	\$9F19
Row 17	40730	40739	\$9F1A	\$9F23
Row 18	40740	40749	\$9F24	\$9F2D
Row 19	40750	40759	\$9F2E	\$9F37
Row 20	40760	40769	\$9F38	\$9F41
Row 21	40770	40779	\$9F42	\$9F4B
Row 22	40780	40789	\$9F4C	\$9F55
Row 23	40790	40799	\$9F56	\$9F5F

**Table E-8. Absolute Addresses for ATARI
BASIC Mode-4 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40320	40329	\$9D80	\$9D89
Row 1	40330	40339	\$9D8A	\$9D93
Row 2	40340	40349	\$9D94	\$9D9D
Row 3	40350	40359	\$9D9E	\$9DA7
Row 4	40360	40369	\$9DA8	\$9DB1
Row 5	40370	40379	\$9DB2	\$9DBB
Row 6	40380	40389	\$9DBC	\$9DC5
Row 7	40390	40399	\$9DC6	\$9DCF
Row 8	40400	40409	\$9DD0	\$9DD9
Row 9	40410	40419	\$9DDA	\$9DE3
Row 10	40420	40429	\$9DE4	\$9DED
Row 11	40430	40439	\$9DEE	\$9DF7
Row 12	40440	40449	\$9DF8	\$9E01
Row 13	40450	40459	\$9E02	\$9E0B
Row 14	40460	40469	\$9E0C	\$9E15
Row 15	40470	40479	\$9E16	\$9E1F
Row 16	40480	40489	\$9E20	\$9E29
Row 17	40490	40499	\$9E2A	\$9E33
Row 18	40500	40509	\$9E34	\$9E3D
Row 19	40510	40519	\$9E3E	\$9E47
Row 20	40520	40529	\$9E48	\$9E51
Row 21	40530	40539	\$9E52	\$9E5B
Row 22	40540	40549	\$9E5C	\$9E65
Row 23	40550	40559	\$9E66	\$9E6F
Row 24	40560	40569	\$9E70	\$9E79
Row 25	40570	40579	\$9E7A	\$9E83
Row 26	40580	40589	\$9E84	\$9E8D
Row 27	40590	40599	\$9E8E	\$9E97
Row 28	40600	40609	\$9E98	\$9EA1
Row 29	40610	40619	\$9EA2	\$9EAB
Row 30	40620	40629	\$9EAC	\$9EB5
Row 31	40630	40639	\$9EB6	\$9EBF
Row 32	40640	40649	\$9EC0	\$9EC9
Row 33	40650	40659	\$9ECA	\$9ED3
Row 34	40660	40669	\$9ED4	\$9EDD
Row 35	40670	40679	\$9EDE	\$9EE7

**Table E-8—cont. Absolute Addresses for ATARI
BASIC Mode-4 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 36	40680	40689	\$9EE8	\$9EF1
Row 37	40690	40699	\$9EF2	\$9EFB
Row 38	40700	40709	\$9EFC	\$9F05
Row 39	40710	40719	\$9F06	\$9F0F
Text window addressing begins here				
Row 0	40800	40839	\$9F60	\$9F87
Row 1	40840	40879	\$9F88	\$9FAF
Row 2	40880	40919	\$9FB0	\$9FD7
Row 3	40920	40959	\$9FD8	\$9FFF

**Table E-9. Absolute Addresses for ATARI
BASIC Mode-20 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	40320	40329	\$9D80	\$9D89
Row 1	40330	40339	\$9D8A	\$9D93
Row 2	40340	40349	\$9D94	\$9D9D
Row 3	40350	40359	\$9D9E	\$9DA7
Row 4	40360	40369	\$9DA8	\$9DB1
Row 5	40370	40379	\$9DB2	\$9DBB
Row 6	40380	40389	\$9DBC	\$9DC5
Row 7	40390	40399	\$9DC6	\$9DCF
Row 8	40400	40409	\$9DD0	\$9DD9
Row 9	40410	40419	\$9DDA	\$9DE3
Row 10	40420	40429	\$9DE4	\$9DED
Row 11	40430	40439	\$9DEE	\$9DF7
Row 12	40440	40449	\$9DF8	\$9E01
Row 13	40450	40459	\$9E02	\$9E0B
Row 14	40460	40469	\$9E0C	\$9E15
Row 15	40470	40479	\$9E16	\$9E1F
Row 16	40480	40489	\$9E20	\$9E29
Row 17	40490	40499	\$9E2A	\$9E33
Row 18	40500	40509	\$9E34	\$9E3D
Row 19	40510	40519	\$9E3E	\$9E47
Row 20	40520	40529	\$9E48	\$9E51
Row 21	40530	40539	\$9E52	\$9E5B
Row 22	40540	40549	\$9E5C	\$9E65
Row 23	40550	40559	\$9E66	\$9E6F
Row 24	40560	40569	\$9E70	\$9E79
Row 25	40570	40579	\$9E7A	\$9E83
Row 26	40580	40589	\$9E84	\$9E8D
Row 27	40590	40599	\$9E8E	\$9E97
Row 28	40600	40609	\$9E98	\$9EA1
Row 29	40610	40619	\$9EA2	\$9EAB
Row 30	40620	40629	\$9EAC	\$9EB5
Row 31	40630	40639	\$9EB6	\$9EBF
Row 32	40640	40649	\$9EC0	\$9EC9
Row 33	40650	40659	\$9ECA	\$9ED3
Row 34	40660	40669	\$9ED4	\$9EDD
Row 35	40670	40679	\$9EDE	\$9EE7

**Table E-9—cont. Absolute Addresses for ATARI
BASIC Mode-20 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 36	40680	40689	\$9EE8	\$9EF1
Row 37	40690	40699	\$9EF2	\$9EFB
Row 38	40700	40709	\$9EFC	\$9F05
Row 39	40710	40719	\$9F06	\$9F0F
Row 40	40720	40729	\$9F10	\$9F19
Row 41	40730	40739	\$9F1A	\$9F23
Row 42	40740	40749	\$9F24	\$9F2D
Row 43	40750	40759	\$9F2E	\$9F37
Row 44	40760	40769	\$9F38	\$9F41
Row 45	40770	40779	\$9F42	\$9F4B
Row 46	40780	40789	\$9F4C	\$9F55
Row 47	40790	40799	\$9F56	\$9F5F

**Table E-10. Absolute Addresses for ATARI
BASIC Mode-5 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	39840	39859	\$9BA0	\$9BB3
Row 1	39860	39879	\$9BB4	\$9BC7
Row 2	39880	39899	\$9BC8	\$9BDB
Row 3	39900	39919	\$9BDC	\$9BEF
Row 4	39920	39939	\$9BF0	\$9C03
Row 5	39940	39959	\$9C04	\$9C17
Row 6	39960	39979	\$9C18	\$9C2B
Row 7	39980	39999	\$9C2C	\$9C3F
Row 8	40000	40019	\$9C40	\$9C53
Row 9	40020	40039	\$9C54	\$9C67
Row 10	40040	40059	\$9C68	\$9C7B
Row 11	40060	40079	\$9C7C	\$9C8F
Row 12	40080	40099	\$9C90	\$9CA3
Row 13	40100	40119	\$9CA4	\$9CB7
Row 14	40120	40139	\$9CB8	\$9CCB
Row 15	40140	40159	\$9CCC	\$9CDF
Row 16	40160	40179	\$9CE0	\$9CF3
Row 17	40180	40199	\$9CF4	\$9D07
Row 18	40200	40219	\$9D08	\$9D1B
Row 19	40220	40239	\$9D1C	\$9D2F
Row 20	40240	40259	\$9D30	\$9D43
Row 21	40260	40279	\$9D44	\$9D57
Row 22	40280	40299	\$9D58	\$9D6B
Row 23	40300	40319	\$9D6C	\$9D7F
Row 24	40320	40339	\$9D80	\$9D93
Row 25	40340	40359	\$9D94	\$9DA7
Row 26	40360	40379	\$9DA8	\$9DBB
Row 27	40380	40399	\$9DBC	\$9DCF
Row 28	40400	40419	\$9DD0	\$9DE3
Row 29	40420	40439	\$9DE4	\$9DF7
Row 30	40440	40459	\$9DF8	\$9E0B
Row 31	40460	40479	\$9E0C	\$9E1F
Row 32	40480	40499	\$9E20	\$9E33
Row 33	40500	40519	\$9E34	\$9E47
Row 34	40520	40539	\$9E48	\$9E5B

**Table E-10—cont. Absolute Addresses for ATARI
BASIC Mode-5 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 35	40540	40559	\$9E5C	\$9E6F
Row 36	40560	40579	\$9E70	\$9E83
Row 37	40580	40599	\$9E84	\$9E97
Row 38	40600	40619	\$9E98	\$9EAB
Row 39	40620	40639	\$9EAC	\$9EBF
Text window addressing begins here				
Row 0	40800	40839	\$9F60	\$9F87
Row 1	40840	40879	\$9F88	\$9FAF
Row 2	40880	40919	\$9FB0	\$9FD7
Row 3	40920	40959	\$9FD8	\$9FFF

**Table E-11. Absolute Addresses for ATARI
BASIC Mode-21 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	39840	39859	\$9BA0	\$9BB3
Row 1	39860	39879	\$9BB4	\$9BC7
Row 2	39880	39899	\$9BC8	\$9BDB
Row 3	39900	39919	\$9BDC	\$9BEF
Row 4	39920	39939	\$9BF0	\$9C03
Row 5	39940	39959	\$9C04	\$9C17
Row 6	39960	39979	\$9C18	\$9C2B
Row 7	39980	39999	\$9C2C	\$9C3F
Row 8	40000	40019	\$9C40	\$9C53
Row 9	40020	40039	\$9C54	\$9C67
Row 10	40040	40059	\$9C68	\$9C7B
Row 11	40060	40079	\$9C7C	\$9C8F
Row 12	40080	40099	\$9C90	\$9CA3
Row 13	40100	40119	\$9CA4	\$9CB7
Row 14	40120	40139	\$9CB8	\$9CCB
Row 15	40140	40159	\$9CCC	\$9CDF
Row 16	40160	40179	\$9CE0	\$9CF3
Row 17	40180	40199	\$9CF4	\$9D07
Row 18	40200	40219	\$9D08	\$9D1B
Row 19	40220	40239	\$9D1C	\$9D2F
Row 20	40240	40259	\$9D30	\$9D43
Row 21	40260	40279	\$9D44	\$9D57
Row 22	40280	40299	\$9D58	\$9D6B
Row 23	40300	40319	\$9D6C	\$9D7F
Row 24	40320	40339	\$9D80	\$9D93
Row 25	40340	40359	\$9D94	\$9DA7
Row 26	40360	40379	\$9DA8	\$9DBB
Row 27	40380	40399	\$9DBC	\$9DCF
Row 28	40400	40419	\$9DD0	\$9DE3
Row 29	40420	40439	\$9DE4	\$9DF7
Row 30	40440	40459	\$9DF8	\$9E0B
Row 31	40460	40479	\$9E0C	\$9E1F
Row 32	40480	40499	\$9E20	\$9E33
Row 33	40500	40519	\$9E34	\$9E47
Row 34	40520	40539	\$9E48	\$9E5B
Row 35	40540	40559	\$9E5C	\$9E6F

**Table E-11—cont. Absolute Addresses for ATARI
BASIC Mode-21 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 36	40560	40579	\$9E70	\$9E83
Row 37	40580	40599	\$9E84	\$9E97
Row 38	40600	40619	\$9E98	\$9EAB
Row 39	40620	40639	\$9EAC	\$9EBF
Row 40	40640	40659	\$9EC0	\$9ED3
Row 41	40660	40679	\$9ED4	\$9EE7
Row 42	40680	40699	\$9EE8	\$9EFB
Row 43	40700	40719	\$9EFC	\$9FOF
Row 44	40720	40739	\$9F10	\$9F23
Row 45	40740	40759	\$9F24	\$9F37
Row 46	40760	40779	\$9F38	\$9F4B
Row 47	40780	40799	\$9F4C	\$9F5F

**Table E-12. Absolute Addresses for ATARI
BASIC Mode-6 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	38880	38899	\$97E0	\$97F3
Row 1	38900	38919	\$97F4	\$9807
Row 2	38920	38939	\$9808	\$981B
Row 3	38940	38959	\$981C	\$982F
Row 4	38960	38979	\$9830	\$9843
Row 5	38980	38999	\$9844	\$9857
Row 6	39000	39019	\$9858	\$986B
Row 7	39020	39039	\$986C	\$987F
Row 8	39040	39059	\$9880	\$9893
Row 9	39060	39079	\$9894	\$98A7
Row 10	39080	39099	\$98A8	\$98BB
Row 11	39100	39119	\$98BC	\$98CF
Row 12	39120	39139	\$98D0	\$98E3
Row 13	39140	39159	\$98E4	\$98F7
Row 14	39160	39179	\$98F8	\$990B
Row 15	39180	39199	\$990C	\$991F
Row 16	39200	39219	\$9920	\$9933
Row 17	39220	39239	\$9934	\$9947
Row 18	39240	39259	\$9948	\$995B
Row 19	39260	39279	\$995C	\$996F
Row 20	39280	39299	\$9970	\$9983
Row 21	39300	39319	\$9984	\$9997
Row 22	39320	39339	\$9998	\$99AB
Row 23	39340	39359	\$99AC	\$99BF
Row 24	39360	39379	\$99C0	\$99D3
Row 25	39380	39399	\$99D4	\$99E7
Row 26	39400	39419	\$99E8	\$99FB
Row 27	39420	39439	\$99FC	\$9A0F
Row 28	39440	39459	\$9A10	\$9A23
Row 29	39460	39479	\$9A24	\$9A37
Row 30	39480	39499	\$9A38	\$9A4B
Row 31	39500	39519	\$9A4C	\$9A5F
Row 32	39520	39539	\$9A60	\$9A73
Row 33	39540	39559	\$9A74	\$9A87
Row 34	39560	39579	\$9A88	\$9A9B
Row 35	39580	39599	\$9A9C	\$9AAF
Row 36	39600	39619	\$9AB0	\$9AC3
Row 37	39620	39639	\$9AC4	\$9AD7
Row 38	39640	39659	\$9AD8	\$9AEB
Row 39	39660	39679	\$9AEC	\$9AFF
Row 40	39680	39699	\$9B00	\$9B13
Row 41	39700	39719	\$9B14	\$9B27
Row 42	39720	39739	\$9B28	\$9B3B
Row 43	39740	39759	\$9B3C	\$9B4F

**Table E-12—cont. Absolute Addresses for ATARI
BASIC Mode-6 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 44	39760	39779	\$9B50	\$9B63
Row 45	39780	39799	\$9B64	\$9B77
Row 46	39800	39819	\$9B78	\$9B8B
Row 47	39820	39839	\$9B8C	\$9B9F
Row 48	39840	39859	\$9BA0	\$9BB3
Row 49	39860	39879	\$9BB4	\$9BC7
Row 50	39880	39899	\$9BC8	\$9BDB
Row 51	39900	39919	\$9BDC	\$9BEF
Row 52	39920	39939	\$9BF0	\$9C03
Row 53	39940	39959	\$9C04	\$9C17
Row 54	39960	39979	\$9C18	\$9C2B
Row 55	39980	39999	\$9C2C	\$9C3F
Row 56	40000	40019	\$9C40	\$9C53
Row 57	40020	40039	\$9C54	\$9C67
Row 58	40040	40059	\$9C68	\$9C7B
Row 59	40060	40079	\$9C7C	\$9C8F
Row 60	40080	40099	\$9C90	\$9CA3
Row 61	40100	40119	\$9CA4	\$9CB7
Row 62	40120	40139	\$9CB8	\$9CCB
Row 63	40140	40159	\$9CCC	\$9CDF
Row 64	40160	40179	\$9CE0	\$9CF3
Row 65	40180	40199	\$9CF4	\$9D07
Row 66	40200	40219	\$9D08	\$9D1B
Row 67	40220	40239	\$9D1C	\$9D2F
Row 68	40240	40259	\$9D30	\$9D43
Row 69	40260	40279	\$9D44	\$9D57
Row 70	40280	40299	\$9D58	\$9D6B
Row 71	40300	40319	\$9D6C	\$9D7F
Row 72	40320	40339	\$9D80	\$9D93
Row 73	40340	40359	\$9D94	\$9DA7
Row 74	40360	40379	\$9DA8	\$9DBB
Row 75	40380	40399	\$9DBC	\$9DCF
Row 76	40400	40419	\$9DD0	\$9DE3
Row 77	40420	40439	\$9DE4	\$9DF7
Row 78	40440	40459	\$9DF8	\$9E0B
Row 79	40460	40479	\$9E0C	\$9E1F
Text window addressing begins here				
Row 0	40800	40839	\$9F60	\$9F87
Row 1	40840	40879	\$9F88	\$9FAF
Row 2	40880	40919	\$9FB0	\$9FD7
Row 3	40920	40959	\$9FD8	\$9FFF

**Table E-13. Absolute Addresses for ATARI
BASIC Mode-22 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 0	38880	38899	\$97E0	\$97F3
Row 1	38900	38919	\$97F4	\$9807
Row 2	38920	38939	\$9808	\$981B
Row 3	38940	38959	\$981C	\$982F
Row 4	38960	38979	\$9830	\$9843
Row 5	38980	38999	\$9844	\$9857
Row 6	39000	39019	\$9858	\$986B
Row 7	39020	39039	\$986C	\$987F
Row 8	39040	39059	\$9880	\$9893
Row 9	39060	39079	\$9894	\$98A7
Row 10	39080	39099	\$98A8	\$98BB
Row 11	39100	39119	\$98BC	\$98CF
Row 12	39120	39139	\$98D0	\$98E3
Row 13	39140	39159	\$98E4	\$98F7
Row 14	39160	39179	\$98F8	\$990B
Row 15	39180	39199	\$990C	\$991F
Row 16	39200	39219	\$9920	\$9933
Row 17	39220	39239	\$9934	\$9947
Row 18	39240	39259	\$9948	\$995B
Row 19	39260	39279	\$995C	\$996F
Row 20	39280	39299	\$9970	\$9983
Row 21	39300	39319	\$9984	\$9997
Row 22	39320	39339	\$9998	\$99AB
Row 23	39340	39359	\$99AC	\$99BF
Row 24	39360	39379	\$99C0	\$99D3
Row 25	39380	39399	\$99D4	\$99E7
Row 26	39400	39419	\$99E8	\$99FB
Row 27	39420	39439	\$99FC	\$9A0F
Row 28	39440	39459	\$9A10	\$9A23
Row 29	39460	39479	\$9A24	\$9A37
Row 30	39480	39499	\$9A38	\$9A4B
Row 31	39500	39519	\$9A4C	\$9A5F
Row 32	39520	39539	\$9A60	\$9A73
Row 33	39540	39559	\$9A74	\$9A87
Row 34	39560	39579	\$9A88	\$9A9B

**Table E-13—cont. Absolute Addresses for ATARI
BASIC Mode-22 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 35	39580	39599	\$9A9C	\$9AAF
Row 36	39600	39619	\$9AB0	\$9AC3
Row 37	39620	39639	\$9AC4	\$9AD7
Row 38	39640	39659	\$9AD8	\$9AEB
Row 39	39660	39679	\$9AEC	\$9AFF
Row 40	39680	39699	\$9B00	\$9B13
Row 41	39700	39719	\$9B14	\$9B27
Row 42	39720	39739	\$9B28	\$9B3B
Row 43	39740	39759	\$9B3C	\$9B4F
Row 44	39760	39779	\$9B50	\$9B63
Row 45	39780	39799	\$9B64	\$9B77
Row 46	39800	39819	\$9B78	\$9B8B
Row 47	39820	39839	\$9B8C	\$9B9F
Row 48	39840	39859	\$9BA0	\$9BB3
Row 49	39860	39879	\$9BB4	\$9BC7
Row 50	39880	39899	\$9BC8	\$9BDB
Row 51	39900	39919	\$9BDC	\$9BEF
Row 52	39920	39939	\$9BF0	\$9C03
Row 53	39940	39959	\$9C04	\$9C17
Row 54	39960	39979	\$9C18	\$9C2B
Row 55	39980	39999	\$9C2C	\$9C3F
Row 56	40000	40019	\$9C40	\$9C53
Row 57	40020	40039	\$9C54	\$9C67
Row 58	40040	40059	\$9C68	\$9C7B
Row 59	40060	40079	\$9C7C	\$9C8F
Row 60	40080	40099	\$9C90	\$9CA3
Row 61	40100	40119	\$9CA4	\$9CB7
Row 62	40120	40139	\$9CB8	\$9CCB
Row 63	40140	40159	\$9CCC	\$9CDF
Row 64	40160	40179	\$9CE0	\$9CF3
Row 65	40180	40199	\$9CF4	\$9D07
Row 66	40200	40219	\$9D08	\$9D1B
Row 67	40220	40239	\$9D1C	\$9D2F
Row 68	40240	40259	\$9D30	\$9D43
Row 69	40260	40279	\$9D44	\$9D57

**Table E-13—cont. Absolute Addresses for ATARI
BASIC Mode-22 Screen Display**

Row	Decimal		Hexadecimal	
	Start	End	Start	End
Row 70	40280	40299	\$9D58	\$9D6B
Row 71	40300	40319	\$9D6C	\$9D7F
Row 72	40320	40339	\$9D80	\$9D93
Row 73	40340	40359	\$9D94	\$9DA7
Row 74	40360	40379	\$9DA8	\$9DBB
Row 75	40380	40399	\$9DBC	\$9DCF
Row 76	40400	40419	\$9DD0	\$9DE3
Row 77	40420	40439	\$9DE4	\$9DF7
Row 78	40440	40459	\$9DF8	\$9E0B
Row 79	40460	40479	\$9E0C	\$9E1F
Row 80	40480	40499	\$9E20	\$9E33
Row 81	40500	40519	\$9E34	\$9E47
Row 82	40520	40539	\$9E48	\$9E5B
Row 83	40540	40559	\$9E5C	\$9E6F
Row 84	40560	40579	\$9E70	\$9E83
Row 85	40580	40599	\$9E84	\$9E97
Row 86	40600	40619	\$9E98	\$9EAB
Row 87	40620	40639	\$9EAC	\$9EBF
Row 88	40640	40659	\$9EC0	\$9ED3
Row 89	40660	40679	\$9ED4	\$9EE7
Row 90	40680	40699	\$9EE8	\$9EFB
Row 91	40700	40719	\$9EFC	\$9F0F
Row 92	40720	40739	\$9F10	\$9F23
Row 93	40740	40759	\$9F24	\$9F37
Row 94	40760	40779	\$9F38	\$9F4B
Row 95	40780	40799	\$9F4C	\$9F5F

**Table E-14. Absolute Addresses for ATARI
BASIC Mode-7 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 0	36960	36999	\$9060	\$9087
Line 1	37000	37039	\$9088	\$90AF
Line 2	37040	37079	\$90B0	\$90D7
Line 3	37080	37119	\$90D8	\$90FF
Line 4	37120	37159	\$9100	\$9127
Line 5	37160	37199	\$9128	\$914F
Line 6	37200	37239	\$9150	\$9177
Line 7	37240	37279	\$9178	\$919F
Line 8	37280	37319	\$91A0	\$91C7
Line 9	37320	37359	\$91C8	\$91EF
Line 10	37360	37399	\$91F0	\$9217
Line 11	37400	37439	\$9218	\$923F
Line 12	37440	37479	\$9240	\$9267
Line 13	37480	37519	\$9268	\$928F
Line 14	37520	37559	\$9290	\$92B7
Line 15	37560	37599	\$92B8	\$92DF
Line 16	37600	37639	\$92E0	\$9307
Line 17	37640	37679	\$9308	\$932F
Line 18	37680	37719	\$9330	\$9E57
Line 19	37720	37759	\$9358	\$937F
Line 20	37760	37799	\$9380	\$93A7
Line 21	37800	37839	\$93A8	\$93CF
Line 22	37840	37879	\$93D0	\$93F7
Line 23	37880	37919	\$93F8	\$941F
Line 24	37920	37959	\$9420	\$9447
Line 25	37960	37999	\$9448	\$946F
Line 26	38000	38039	\$9470	\$9497
Line 27	38040	38079	\$9498	\$94BF
Line 28	38080	38119	\$94C0	\$94E7
Line 29	38120	38159	\$94E8	\$950F
Line 30	38160	38199	\$9510	\$9537
Line 31	38200	38239	\$9538	\$955F
Line 32	38240	38279	\$9560	\$9587
Line 33	38280	38319	\$9588	\$95AF
Line 34	38320	38359	\$95B0	\$95D7
Line 35	38360	38399	\$95D8	\$95FF
Line 36	38400	38439	\$9600	\$9627
Line 37	38440	38479	\$9628	\$964F
Line 38	38480	38519	\$9650	\$9677
Line 39	38520	38559	\$9678	\$969F
Line 40	38560	38599	\$96A0	\$96C7
Line 41	38600	38639	\$96C8	\$96EF
Line 42	38640	38679	\$96F0	\$9717
Line 43	38680	38719	\$9718	\$973F

**Table E-14—cont. Absolute Addresses for ATARI
BASIC Mode-7 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 44	38720	38759	\$9740	\$9767
Line 45	38760	38799	\$9768	\$978F
Line 46	38800	38839	\$9790	\$97B7
Line 47	38840	38879	\$97B8	\$97DF
Line 48	38880	38919	\$97E0	\$9807
Line 49	38920	38959	\$9808	\$982F
Line 50	38960	38999	\$9830	\$9857
Line 51	39000	39039	\$9858	\$987F
Line 52	39040	39079	\$9880	\$98A7
Line 53	39080	39119	\$98A8	\$98CF
Line 54	39120	39159	\$98D0	\$98F7
Line 55	39160	39199	\$98F8	\$991F
Line 56	39200	39239	\$9920	\$9947
Line 57	39240	39279	\$9948	\$996F
Line 58	39280	39319	\$9970	\$9997
Line 59	39320	39359	\$9998	\$99BF
Line 60	39360	39399	\$99C0	\$99E7
Line 61	39400	39439	\$99E8	\$9A0F
Line 62	39440	39479	\$9A10	\$9A37
Line 63	39480	39519	\$9A38	\$9A5F
Line 64	39520	39559	\$9A60	\$9A87
Line 65	39560	39599	\$9A88	\$9AAF
Line 66	39600	39639	\$9AB0	\$9AD7
Line 67	39640	39679	\$9AD8	\$9AFF
Line 68	39680	39719	\$9B00	\$9B27
Line 69	39720	39759	\$9B28	\$9B4F
Line 70	39760	39799	\$9B50	\$9B77
Line 71	39800	39839	\$9B78	\$9B9F
Line 72	39840	39879	\$9BA0	\$9BC7
Line 73	39880	39919	\$9BC8	\$9BEF
Line 74	39920	39959	\$9BF0	\$9C17
Line 75	39960	39999	\$9C18	\$9C3F
Line 76	40000	40039	\$9C40	\$9C67
Line 77	40040	40079	\$9C68	\$9C8F
Line 78	40080	40119	\$9C90	\$9CB7
Line 79	40120	40159	\$9CB8	\$9CDF
Text window addressing begins here				
Line 0	40800	40839	\$9F60	\$9F87
Line 1	40840	40879	\$9F88	\$9FAF
Line 2	40880	40919	\$9FB0	\$9FD7
Line 3	40920	40959	\$9FD8	\$9FFF

**Table E-15. Absolute Addresses for ATARI
BASIC Mode-23 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 0	36960	36999	\$9060	\$9087
Line 1	37000	37039	\$9088	\$90AF
Line 2	37040	37079	\$90B0	\$90D7
Line 3	37080	37119	\$90D8	\$90FF
Line 4	37120	37159	\$9100	\$9127
Line 5	37160	37199	\$9128	\$914F
Line 6	37200	37239	\$9150	\$9177
Line 7	37240	37279	\$9178	\$919F
Line 8	37280	37319	\$91A0	\$91C7
Line 9	37320	37359	\$91C8	\$91EF
Line 10	37360	37399	\$91F0	\$9217
Line 11	37400	37439	\$9218	\$923F
Line 12	37440	37479	\$9240	\$9267
Line 13	37480	37519	\$9268	\$928F
Line 14	37520	37559	\$9290	\$92B7
Line 15	37560	37599	\$92B8	\$92DF
Line 16	37600	37639	\$92E0	\$9307
Line 17	37640	37679	\$9308	\$932F
Line 18	37680	37719	\$9330	\$9357
Line 19	37720	37759	\$9358	\$937F
Line 20	37760	37799	\$9380	\$93A7
Line 21	37800	37839	\$93A8	\$93CF
Line 22	37840	37879	\$93D0	\$93F7
Line 23	37880	37919	\$93F8	\$941F
Line 24	37920	37959	\$9420	\$9447
Line 25	37960	37999	\$9448	\$946F
Line 26	38000	38039	\$9470	\$9497
Line 27	38040	38079	\$9498	\$94BF
Line 28	38080	38119	\$94C0	\$94E7
Line 29	38120	38159	\$94E8	\$950F
Line 30	38160	38199	\$9510	\$9537
Line 31	38200	38239	\$9538	\$955F
Line 32	38240	38279	\$9560	\$9587
Line 33	38280	38319	\$9588	\$95AF
Line 34	38320	38359	\$95B0	\$95D7

**Table E-15—cont. Absolute Addresses for ATARI
BASIC Mode-23 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 35	38360	38399	\$95D8	\$95FF
Line 36	38400	38439	\$9600	\$9627
Line 37	38440	38479	\$9628	\$964F
Line 38	38480	38519	\$9650	\$9677
Line 39	38520	38559	\$9678	\$969F
Line 40	38560	38599	\$96A0	\$96C7
Line 41	38600	38639	\$96C8	\$96EF
Line 42	38640	38679	\$96F0	\$9717
Line 43	38680	38719	\$9718	\$973F
Line 44	38720	38759	\$9740	\$9767
Line 45	38760	38799	\$9768	\$978F
Line 46	38800	38839	\$9790	\$97B7
Line 47	38840	38879	\$97B8	\$97DF
Line 48	38880	38919	\$97E0	\$9807
Line 49	38920	38959	\$9808	\$982F
Line 50	38960	38999	\$9830	\$9857
Line 51	39000	39039	\$9858	\$987F
Line 52	39040	39079	\$9880	\$98A7
Line 53	39080	39119	\$98A8	\$98CF
Line 54	39120	39159	\$98D0	\$98F7
Line 55	39160	39199	\$98F8	\$991F
Line 56	39200	39239	\$9920	\$9947
Line 57	39240	39279	\$9948	\$996F
Line 58	39280	39319	\$9970	\$9997
Line 59	39320	39359	\$9998	\$99BF
Line 60	39360	39399	\$99C0	\$99E7
Line 61	39400	39439	\$99E8	\$9A0F
Line 62	39440	39479	\$9A10	\$9A37
Line 63	39480	39519	\$9A38	\$9A5F
Line 64	39520	39559	\$9A60	\$9A87
Line 65	39560	39599	\$9A88	\$9AAF
Line 66	39600	39639	\$9AB0	\$9AD7
Line 67	39640	39679	\$9AD8	\$9AFF
Line 68	39680	39719	\$9B00	\$9B27
Line 69	39720	39759	\$9B28	\$9B4F

**Table E-15—cont. Absolute Addresses for ATARI
BASIC Mode-23 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 70	39760	39799	\$9B50	\$9B77
Line 71	39800	39839	\$9B78	\$9B9F
Line 72	39840	39879	\$9BA0	\$9BC7
Line 73	39880	39919	\$9BC8	\$9BEF
Line 74	39920	39959	\$9BF0	\$9C17
Line 75	39960	39999	\$9C18	\$9C3F
Line 76	40000	40039	\$9C40	\$9C67
Line 77	40040	40079	\$9C68	\$9C8F
Line 78	40080	40119	\$9C90	\$9CB7
Line 79	40120	40159	\$9CB8	\$9CDF
Line 80	40160	40199	\$9CE0	\$9D07
Line 81	40200	40239	\$9D08	\$9D2F
Line 82	40240	40279	\$9D30	\$9D57
Line 83	40280	40319	\$9D58	\$9D7F
Line 84	40320	40359	\$9D80	\$9DA7
Line 85	40360	40399	\$9DA8	\$9DCF
Line 86	40400	40439	\$9DD0	\$9DF7
Line 87	40440	40479	\$9DF8	\$9E1F
Line 88	40480	40519	\$9E20	\$9E47
Line 89	40520	40559	\$9E48	\$9E6F
Line 90	40560	40599	\$9E70	\$9E97
Line 91	40600	40639	\$9E98	\$9EBF
Line 92	40640	40679	\$9EC0	\$9EE7
Line 93	40680	40719	\$9EE8	\$9F0F
Line 94	40720	40759	\$9F10	\$9F37
Line 95	40760	40799	\$9F38	\$9F5F

**Table E-16. Absolute Addresses for ATARI
BASIC Mode-8 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 0	33104	33143	\$8150	\$8177
Line 1	33144	33183	\$8178	\$819F
Line 2	33184	33223	\$81A0	\$81C7
Line 3	33224	33263	\$81C8	\$81EF
Line 4	33264	33303	\$81F0	\$8217
Line 5	33304	33343	\$8218	\$823F
Line 6	33344	33383	\$8240	\$8267
Line 7	33384	33423	\$8268	\$828F
Line 8	33424	33463	\$8290	\$82B7
Line 9	33464	33503	\$82B8	\$82DF
Line 10	33504	33543	\$82E0	\$8307
Line 11	33544	33583	\$8308	\$832F
Line 12	33584	33623	\$8330	\$8357
Line 13	33624	33663	\$8358	\$837F
Line 14	33664	33703	\$8380	\$83A7
Line 15	33704	33743	\$83A8	\$83CF
Line 16	33744	33783	\$83D0	\$83F7
Line 17	33784	33823	\$83F8	\$841F
Line 18	33824	33863	\$8420	\$8447
Line 19	33864	33903	\$8448	\$846F
Line 20	33904	33943	\$8470	\$8497
Line 21	33944	33983	\$8498	\$84BF
Line 22	33984	34023	\$84C0	\$84E7
Line 23	34024	34063	\$84E8	\$850F
Line 24	34064	34103	\$8510	\$8537
Line 25	34104	34143	\$8538	\$855F
Line 26	34144	34183	\$8560	\$8587
Line 27	34184	34223	\$8588	\$85AF
Line 28	34224	34263	\$85B0	\$85D7
Line 29	34264	34303	\$85D8	\$85FF
Line 30	34304	34343	\$8600	\$8627
Line 31	34344	34383	\$8628	\$864F
Line 32	34384	34423	\$8650	\$8677
Line 33	34424	34463	\$8678	\$869F
Line 34	34464	34503	\$86A0	\$86C7
Line 35	34504	34543	\$86C8	\$86EF
Line 36	34544	34583	\$86F0	\$8717
Line 37	34584	34623	\$8718	\$873F
Line 38	34624	34663	\$8740	\$8767
Line 39	34664	34703	\$8768	\$878F
Line 40	34704	34743	\$8790	\$87B7
Line 41	34744	34783	\$87B8	\$87DF

**Table E-16—cont. Absolute Addresses for ATARI
BASIC Mode-8 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 42	34784	34823	\$87E0	\$8807
Line 43	34824	34863	\$8808	\$882F
Line 44	34864	34903	\$8830	\$8857
Line 45	34904	34943	\$8858	\$887F
Line 46	34944	34983	\$8880	\$88A7
Line 47	34984	35023	\$88A8	\$88CF
Line 48	35024	35063	\$88D0	\$88F7
Line 49	35064	35103	\$88F8	\$891F
Line 50	35104	35243	\$8920	\$8947
Line 51	35144	35183	\$8948	\$896F
Line 52	35184	35223	\$8970	\$8997
Line 53	35224	35263	\$8998	\$89BF
Line 54	35264	35303	\$89C0	\$89E7
Line 55	35304	35343	\$89E8	\$8A0F
Line 56	35344	35383	\$8A10	\$8A37
Line 57	35384	35423	\$8A38	\$8A5F
Line 58	35424	35463	\$8A60	\$8A87
Line 59	35464	35503	\$8A88	\$8AAF
Line 60	35504	35543	\$8AB0	\$8AD7
Line 61	35544	35583	\$8AD8	\$8AFF
Line 62	35584	35623	\$8B00	\$8B27
Line 63	35624	35663	\$8B28	\$8B4F
Line 64	35664	35703	\$8B50	\$8B77
Line 65	35704	35743	\$8B78	\$8B9F
Line 66	35744	35783	\$8BA0	\$8BC7
Line 67	35784	35823	\$8BC8	\$8BEF
Line 68	35824	35863	\$8BF0	\$8C17
Line 69	35864	35903	\$8C18	\$8C3F
Line 70	35904	35943	\$8C40	\$8C67
Line 71	35944	35983	\$8C68	\$8C8F
Line 72	35984	36023	\$8C90	\$8CB7
Line 73	36024	36063	\$8CB8	\$8CDF
Line 74	36064	36103	\$8CE0	\$8D07
Line 75	36104	36143	\$8D08	\$8D2F
Line 76	36144	36183	\$8D30	\$8D57
Line 77	36184	36223	\$8D58	\$8D7F
Line 78	36224	36263	\$8D80	\$8DA7
Line 79	36264	36303	\$8DA8	\$8DCF
Line 80	36304	36343	\$8DD0	\$8DF7
Line 81	36344	36383	\$8DF8	\$8E1F
Line 82	36384	36423	\$8E20	\$8E47
Line 83	36424	36463	\$8E48	\$8E6F

**Table E-16—cont. Absolute Addresses for ATARI
BASIC Mode-8 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 84	36464	36503	\$8E70	\$8E97
Line 85	36504	36543	\$8E98	\$8EBF
Line 86	36544	36583	\$8EC0	\$8EE7
Line 87	36584	36623	\$8EE8	\$8F0F
Line 88	36624	36663	\$8F10	\$8F37
Line 89	36664	36703	\$8F38	\$8F5F
Line 90	36704	36743	\$8F60	\$8F87
Line 91	36744	36783	\$8F88	\$8FAF
Line 92	36784	36823	\$8FB0	\$8FD7
Line 93	36824	36863	\$8FD8	\$8FFF
Line 94	36864	36903	\$9000	\$9027
Line 95	36904	36943	\$9028	\$904F
Line 96	36944	36983	\$9050	\$9077
Line 97	36984	37023	\$9078	\$909F
Line 98	37024	37063	\$90A0	\$90C7
Line 99	37064	37103	\$90C8	\$90EF
Line 100	37104	37143	\$90F0	\$9117
Line 101	37144	37183	\$9118	\$913F
Line 102	37184	37223	\$9140	\$9167
Line 103	37224	37263	\$9168	\$918F
Line 104	37264	37303	\$9190	\$91B7
Line 105	37304	37343	\$91B8	\$91DF
Line 106	37344	37383	\$91E0	\$9207
Line 107	37384	37423	\$9208	\$922F
Line 108	37424	37463	\$9230	\$9257
Line 109	37464	37503	\$9258	\$927F
Line 110	37504	37543	\$9280	\$92A7
Line 111	37544	37583	\$92A8	\$92CF
Line 112	37584	37623	\$92D0	\$92F7
Line 113	37624	37663	\$92F8	\$931F
Line 114	37664	37703	\$9320	\$9347
Line 115	37704	37743	\$9348	\$936F
Line 116	37744	37783	\$9370	\$9397
Line 117	37784	37823	\$9398	\$93BF
Line 118	37824	37863	\$93C0	\$93E7
Line 119	37864	37903	\$93E8	\$940F
Line 120	37904	37943	\$9410	\$9437
Line 121	37944	37983	\$9438	\$945F
Line 122	37984	38023	\$9460	\$9487
Line 123	38024	38063	\$9488	\$94AF
Line 124	38064	38103	\$94B0	\$94D7
Line 125	38104	38143	\$94D8	\$94FF

**Table E-16—cont. Absolute Addresses for ATARI
BASIC Mode-8 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 126	38144	38183	\$9500	\$9527
Line 127	38184	38223	\$9528	\$954F
Line 128	38224	38263	\$9550	\$9577
Line 129	38264	38303	\$9578	\$959F
Line 130	38304	38343	\$95A0	\$95C7
Line 131	38344	38383	\$95C8	\$95EF
Line 132	38384	38423	\$95F0	\$9617
Line 133	38424	38463	\$9618	\$963F
Line 134	38464	38503	\$9640	\$9667
Line 135	38504	38543	\$9668	\$968F
Line 136	38544	38583	\$9690	\$96B7
Line 137	38584	38623	\$96B8	\$96DF
Line 138	38624	38663	\$96E0	\$9707
Line 139	38664	38703	\$9708	\$972F
Line 140	38704	38743	\$9730	\$9757
Line 141	38744	38783	\$9758	\$977F
Line 142	38784	38823	\$9780	\$97A7
Line 143	38824	38863	\$97A8	\$97CF
Line 144	38864	38903	\$97D0	\$97F7
Line 145	38904	38943	\$97F8	\$981F
Line 146	38944	38983	\$9820	\$9847
Line 147	38984	39023	\$9848	\$986F
Line 148	39024	39063	\$9870	\$9897
Line 149	39064	39103	\$9898	\$98BF
Line 150	39104	39143	\$98C0	\$98E7
Line 151	39144	39183	\$98E8	\$990F
Line 152	39184	39223	\$9910	\$9937
Line 153	39224	39263	\$9938	\$995F
Line 154	39264	39303	\$9960	\$9987
Line 155	39304	39343	\$9988	\$99AF
Line 156	39344	39383	\$99B0	\$99D7
Line 157	39384	39423	\$99D8	\$99FF
Line 158	39424	39463	\$9A00	\$9A27
Line 159	39464	39503	\$9A28	\$9A4F
Text window addressing begins here				
Line 0	40800	40839	\$9F60	\$9F87
Line 1	40840	40879	\$9F88	\$9FAF
Line 2	40880	40919	\$9FB0	\$9FD7
Line 3	40920	40959	\$9FD8	\$9FFF

**Table E-17. Absolute Addresses for ATARI
BASIC Mode-24 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 0	33104	33143	\$8150	\$8177
Line 1	33144	33183	\$8178	\$819F
Line 2	33184	33223	\$81A0	\$81C7
Line 3	33224	33263	\$81C8	\$81EF
Line 4	33264	33303	\$81F0	\$8217
Line 5	33304	33343	\$8218	\$823F
Line 6	33344	33383	\$8240	\$8267
Line 7	33384	33423	\$8268	\$828F
Line 8	33424	33463	\$8290	\$82B7
Line 9	33464	33503	\$82B8	\$82DF
Line 10	33504	33543	\$82E0	\$8307
Line 11	33544	33583	\$8308	\$832F
Line 12	33584	33623	\$8330	\$8357
Line 13	33624	33663	\$8358	\$837F
Line 14	33664	33703	\$8380	\$83A7
Line 15	33704	33743	\$83A8	\$83CF
Line 16	33744	33783	\$83D0	\$83F7
Line 17	33784	33823	\$83F8	\$841F
Line 18	33824	33863	\$8420	\$8447
Line 19	33864	33903	\$8448	\$846F
Line 20	33904	33943	\$8470	\$8497
Line 21	33944	33983	\$8498	\$84BF
Line 22	33984	34023	\$84C0	\$84E7
Line 23	34024	34063	\$84E8	\$850F
Line 24	34064	34103	\$8510	\$8537
Line 25	34104	34143	\$8538	\$855F
Line 26	34144	34183	\$8560	\$8587
Line 27	34184	34223	\$8588	\$85AF
Line 28	34224	34263	\$85B0	\$85D7
Line 29	34264	34303	\$85D8	\$85FF
Line 30	34304	34343	\$8600	\$8627
Line 31	34344	34383	\$8628	\$864F
Line 32	34384	34423	\$8650	\$8677
Line 33	34424	34463	\$8678	\$869F
Line 34	34464	34503	\$86A0	\$86C7
Line 35	34504	34543	\$86C8	\$86EF
Line 36	34544	34583	\$86F0	\$8717
Line 37	34584	34623	\$8718	\$873F
Line 38	34624	34663	\$8740	\$8767

**Table E-17—cont. Absolute Addresses for ATARI
BASIC Mode-24 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 39	34664	34703	\$8768	\$878F
Line 40	34704	34743	\$8790	\$87B7
Line 41	34744	34783	\$87B8	\$87DF
Line 42	34784	34823	\$87E0	\$8807
Line 43	34824	34863	\$8808	\$882F
Line 44	34864	34903	\$8830	\$8857
Line 45	34904	34943	\$8858	\$887F
Line 46	34944	34983	\$8880	\$88A7
Line 47	34984	35023	\$88A8	\$88CF
Line 48	35024	35063	\$88D0	\$88F7
Line 49	35064	35103	\$88F8	\$891F
Line 50	35104	35143	\$8920	\$8947
Line 51	35144	35183	\$8948	\$896F
Line 52	35184	35223	\$8970	\$8997
Line 53	35224	35263	\$8998	\$89BF
Line 54	35264	35303	\$89C0	\$89E7
Line 55	35304	35343	\$89E8	\$8A0F
Line 56	35344	35383	\$8A10	\$8A37
Line 57	35384	35423	\$8A38	\$8A5F
Line 58	35424	35463	\$8A60	\$8A87
Line 59	35464	35503	\$8A88	\$8AAF
Line 60	35504	35543	\$8AB0	\$8AD7
Line 61	35544	35583	\$8AD8	\$8AFF
Line 62	35584	35623	\$8B00	\$8B27
Line 63	35624	35663	\$8B28	\$8B4F
Line 64	35664	35703	\$8B50	\$8B77
Line 65	35704	35743	\$8B78	\$8B9F
Line 66	35744	35783	\$8BA0	\$8BC7
Line 67	35784	35823	\$8BC8	\$8BEF
Line 68	35824	35863	\$8BF0	\$8C17
Line 69	35864	35903	\$8C18	\$8C3F
Line 70	35904	35943	\$8C40	\$8C67
Line 71	35944	35983	\$8C68	\$8C8F
Line 72	35984	36023	\$8C90	\$8CB7
Line 73	36024	36063	\$8CB8	\$8CDF
Line 74	36064	36103	\$8CE0	\$8D07
Line 75	36104	36143	\$8D08	\$8D2F
Line 76	36144	36183	\$8D30	\$8D57
Line 77	36184	36223	\$8D58	\$8D7F

**Table E-17—cont. Absolute Addresses for ATARI
BASIC Mode-24 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 78	36224	36263	\$8D80	\$8DA7
Line 79	36264	36303	\$8DA8	\$8DCF
Line 80	36304	36343	\$8DD0	\$8DF7
Line 81	36344	36383	\$8DF8	\$8E1F
Line 82	36384	36423	\$8E20	\$8E47
Line 83	36424	36463	\$8E48	\$8E6F
Line 84	36464	36503	\$8E70	\$8E97
Line 85	36504	36543	\$8E98	\$8EBF
Line 86	36544	36583	\$8EC0	\$8EE7
Line 87	36584	36623	\$8EE8	\$8F0F
Line 88	36624	36663	\$8F10	\$8F37
Line 89	36664	36703	\$8F38	\$8F5F
Line 90	36704	36743	\$8F60	\$8F87
Line 91	36744	36783	\$8F88	\$8FAF
Line 92	36784	36823	\$8FB0	\$8FD7
Line 93	36824	36863	\$8FD8	\$8FFF
Line 94	36864	36903	\$9000	\$9027
Line 95	36904	36943	\$9028	\$904F
Line 96	36944	36983	\$9050	\$9077
Line 97	36984	37023	\$9078	\$909F
Line 98	37024	37063	\$90A0	\$90C7
Line 99	37064	37103	\$90C8	\$90EF
Line 100	37104	37143	\$90F0	\$9117
Line 101	37144	37183	\$9118	\$913F
Line 102	37184	37223	\$9140	\$9167
Line 103	37224	37263	\$9168	\$918F
Line 104	37264	37303	\$9190	\$91B7
Line 105	37304	37343	\$91B8	\$91DF
Line 106	37344	37383	\$91E0	\$9207
Line 107	37384	37423	\$9208	\$922F
Line 108	37424	37463	\$9230	\$9257
Line 109	37464	37503	\$9258	\$927F
Line 110	37504	37543	\$9280	\$92A7
Line 111	37544	37583	\$92A8	\$92CF
Line 112	37584	37623	\$92D0	\$92F7
Line 113	37624	37663	\$92F8	\$931F
Line 114	37664	37703	\$9320	\$9347
Line 115	37704	37743	\$9348	\$936F

**Table E-17—cont. Absolute Addresses for ATARI
BASIC Mode-24 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 116	37744	37783	\$9370	\$9397
Line 117	37784	37823	\$9398	\$93BF
Line 118	37824	37863	\$93C0	\$93E7
Line 119	37864	37903	\$93E8	\$940F
Line 120	37904	37943	\$9410	\$9437
Line 121	37944	37983	\$9438	\$945F
Line 122	37984	38023	\$9460	\$9487
Line 123	38024	38063	\$9488	\$94AF
Line 124	38064	38103	\$94B0	\$94D7
Line 125	38104	38143	\$94D8	\$94FF
Line 126	38144	38183	\$9500	\$9527
Line 127	38184	38223	\$9528	\$954F
Line 128	38224	38263	\$9550	\$9577
Line 129	38264	38303	\$9578	\$959F
Line 130	38304	38343	\$95A0	\$95C7
Line 131	38344	38383	\$95C8	\$95EF
Line 132	38384	38423	\$95F0	\$9617
Line 133	38424	38463	\$9618	\$963F
Line 134	38464	38503	\$9640	\$9667
Line 135	38504	38543	\$9668	\$968F
Line 136	38544	38583	\$9690	\$96B7
Line 137	38584	38623	\$96B8	\$96DF
Line 138	38624	38663	\$96E0	\$9707
Line 139	38664	38703	\$9708	\$972F
Line 140	38704	38743	\$9730	\$9757
Line 141	38744	38783	\$9758	\$977F
Line 142	38784	38823	\$9780	\$97A7
Line 143	38824	38863	\$97A8	\$97CF
Line 144	38864	38903	\$97D0	\$97F7
Line 145	38904	38943	\$97F8	\$981F
Line 146	38944	38983	\$9820	\$9847
Line 147	38984	39023	\$9848	\$986F
Line 148	39024	39063	\$9870	\$9897
Line 149	39064	39103	\$9898	\$98BF
Line 150	39104	39143	\$98C0	\$98E7
Line 151	39144	39183	\$98E8	\$990F
Line 152	39184	39223	\$9910	\$9937
Line 153	39224	39263	\$9938	\$995F

**Table E-17—cont. Absolute Addresses for ATARI
BASIC Mode-24 Screen Display**

Line	Decimal		Hexadecimal	
	Start	End	Start	End
Line 154	39264	39303	\$9960	\$9987
Line 155	39304	39343	\$9988	\$99AF
Line 156	39344	39383	\$99B0	\$99D7
Line 157	39384	39423	\$99D8	\$99FF
Line 158	39424	39463	\$9A00	\$9A27
Line 159	39464	39503	\$9A28	\$9A4F
Line 160	39504	39543	\$9A50	\$9A77
Line 161	39544	39583	\$9A78	\$9A9F
Line 162	39584	39623	\$9AA0	\$9AC7
Line 163	39624	39663	\$9AC8	\$9AEF
Line 164	39664	39703	\$9AF0	\$9B17
Line 165	39704	39743	\$9B18	\$9B3F
Line 166	39744	39783	\$9B40	\$9B67
Line 167	39784	39823	\$9B68	\$9B8F
Line 168	39824	39863	\$9B90	\$9BB7
Line 169	39864	39903	\$9BB8	\$9BDF
Line 170	39904	39943	\$9BE0	\$9C07
Line 171	39944	39983	\$9C08	\$9C2F
Line 172	39984	40023	\$9C30	\$9C57
Line 173	40024	40063	\$9C58	\$9C7F
Line 174	40064	40103	\$9C80	\$9CA7
Line 175	40104	40143	\$9CA8	\$9CCF
Line 176	40144	40183	\$9CD0	\$9CF7
Line 177	40184	40223	\$9CF8	\$9D1F
Line 178	40224	40263	\$9D20	\$9D47
Line 179	40264	40303	\$9D48	\$9D6F
Line 180	40304	40343	\$9D70	\$9D97
Line 181	40344	40383	\$9D98	\$9DBF
Line 182	40384	40423	\$9DC0	\$9DE7
Line 183	40424	40463	\$9DE8	\$9E0F
Line 184	40464	40503	\$9E10	\$9E37
Line 185	40504	40543	\$9E38	\$9E5F
Line 186	40544	40583	\$9E60	\$9E87
Line 187	40584	40623	\$9E88	\$9EAF
Line 188	40624	40663	\$9EB0	\$9ED7
Line 189	40664	40703	\$9ED8	\$9EFF
Line 190	40704	40743	\$9F00	\$9F27
Line 191	40744	40783	\$9F28	\$9F4F

Appendix F

Derived Trigonometric Functions

The following functions are not represented in ATARI BASIC. They can be executed, however, by applying the corresponding equivalent expressions in Table F-1.

Table F-1. Derived Trigonometric Functions

Function	Equivalent
Secant	$1/\text{COS}(X)$
Cosecant	$1/\text{SIN}(X)$
Cotangent	$1/\text{TAN}(X)$
Inverse sine	$\text{ATN}(X/\text{SQR}(1-X^2))$
Inverse cosine	$k - \text{ATN}(X/\text{SQR}(1-X^2))$
Inverse secant	$\text{ATN}(\text{SQR}(X^2-1)) + \text{SGN}(X-1)*k$
Inverse cosecant	$\text{ATN}(1/\text{SQR}(X^2-1)) + \text{SGN}(X-1) + k$
Inverse cotangent	$\text{ATN}(X) + k$
Hyperbolic sine	$(\text{EXP}(X) - \text{EXP}(-X))/2$
Hyperbolic cosine	$(\text{EXP}(X) + \text{EXP}(-X))/2$
Hyperbolic tangent	$(\text{EXP}(X) - \text{EXP}(-X))/(\text{EXP}(X) + \text{EXP}(-X))$
Hyperbolic secant	$2/(\text{EXP}(X) + \text{EXP}(-X))$
Hyperbolic cosecant	$2/(\text{EXP}(X) - \text{EXP}(-X))$
Hyperbolic cotangent	$(\text{EXP}(X) + \text{EXP}(-X))/(\text{EXP}(X) - \text{EXP}(-X))$
Inverse hyperbolic sine	$\text{LOG}(X + \text{SQR}(X^2+1))$
Inverse hyperbolic cosine	$\text{LOG}(X + \text{SQR}(X^2-1))$
Inverse hyperbolic tangent	$\text{LOG}((1+X)/(1-X))/2$
Inverse hyperbolic secant	$\text{LOG}((1 - \text{SQR}(1-X^2))/X)$
Inverse hyperbolic cosecant	$\text{LOG}((1 + \text{SGN}(X)*\text{SQR}(1+X^2))/X)$
Inverse hyperbolic cotangent	$\text{LOG}(X+1)/(X-1)/2$

NOTE: The value assigned to variable k in these functions should be 1.570796 if in the RAD (radian) mode, or 90 if in the DEG (degree) mode.

Appendix G

ATARI Error and Status Codes

The ATARI operating system does not generate error messages in a plain-text format. Rather, it displays an error code number. Table G-1 lists the error code numbers and their general interpretations.

The current status of a given IOCB operation can be checked by executing a `STATUS #chan,x` statement; where *chan* is the IOCB channel to be tested, and *x* is a numeric variable that takes on the status code. It is also possible to see the current status for all eight IOCB channels by `PEEKING` or loading the 6502 from the following RAM locations:

IOCB Channel	Status Register Address	
	Dec	Hex
0	833	\$0343
1	851	\$0353
2	867	\$0363
3	883	\$0373
4	899	\$0383
5	915	\$0393
6	931	\$03A3
7	947	\$03B3

Table G-2 shows the status codes (both decimal and hexadecimal) and their meaning.

Table G-1. ATARI Error Codes and Their Meaning

Number	Meaning
2	<p>Out of memory.</p> <p>There is insufficient RAM available for the size of the program or programming task.</p>
3	<p>Illegal function call.</p> <p>A numerical value is too large, too small or has a sign that is inappropriate for the statement that uses it.</p>
4	<p>Too many variables.</p> <p>ATARI BASIC allows no more than 128 different variable names to be used throughout the execution of a program.</p>
5	<p>String too long.</p> <p>Actual string assignment contains more characters than DIMensioned its variable name. The ERROR message does not appear on all ATARI systems; the string is simply truncated to the DIMensioned length, and no interruption occurs.</p>
6	<p>Out of DATA error.</p> <p>The number of READ operations executed by the program exceeds the number of items in the corresponding DATA list.</p>
7	<p>Number exceeds 32767.</p> <p>This error indicates that an operation that is expecting a positive integer value is getting either a value that exceeds 32767 or is a negative value.</p>
8	<p>Type mismatch in an INPUT statement.</p> <p>An INPUT statement that assigns values to a numerical variable cannot accept a string value from the keyboard.</p>
9	<p>DIMension error.</p> <p>This error occurs when attempting to use a non-DIMensioned string or array variable, to DIMension the same string or array variable more than one time, or to DIMension an array beyond the limit of 32767 elements.</p>
10	<p>Expression too complex.</p> <p>A string or arithmetic expression is too complex, an arithmetic expression contains too many levels of parentheses nesting, or the program contains too many nested GOSUB statements.</p>
11	<p>Floating-point value overflow/underflow.</p> <p>The result of a floating-point arithmetic operation yielded a value that exceeds the system's notation format. This error occurs, for instance, when attempting to divide by zero.</p>

Table G-1—cont. ATARI Error Codes and Their Meaning

Number	Meaning
12	Invalid line number. A GOSUB or GOTO statement references a line number that does not exist in the program.
13	NEXT without FOR. The program contains a NEXT statement that does not have a corresponding FOR statement.
14	Line too long. A BASIC line contains statements that are too complex or too long.
15	Missing GOSUB or FOR. The program contains RETURN or NEXT statements whose corresponding GOSUB or FOR was deleted since the execution of the last RUN command.
16	RETURN without GOSUB. The program uses a RETURN statement that does not have a corresponding GOSUB.
17	Meaningless RAM-byte error. Data previously POKEd into RAM does not make sense when read during the execution of a program. Possibly a hardware fault, but most often the result of POKEing invalid data into RAM.
18	Invalid string character. A nonnumeric character resides within a string when the program attempts to execute a VAL function on that string.
19	LOAD program too large for RAM. A program being loaded from cassette or disk is too long for the amount of RAM that is available in the system.
20	Bad device number. A specified device number is equal to 0 or greater than 7.
21	File not in LOAD format. The program is using a LOAD statement for files or programs saved with a format other than SAVE.
128	BREAK abort. The user aborted an I/O operation by striking the BREAK key.
129	Device already OPEN. The program attempts to OPEN a device channel that is already OPEN.

Table G-1—cont. ATARI Error Codes and Their Meaning

Number	Meaning
130	Bad device specification. The program specifies an unrecognized device code, or the specified device is not properly connected to the system.
131	Write-only error. The program specifies a read operation from a write-only device (such as a printer).
132	Bad XIO syntax. An XIO statement is using inappropriate or unintelligible specifications.
133	Not-OPEN error. Program attempts to use a device or file that was not previously OPENed.
134	Bad I/O channel number. Program attempts to use a device number other than 1, 2, 3, 4, 5, 6, or 7.
135	Read-only error. The program attempts to write to a device or file that is specified for reading operations.
136	EOF error. The program has found the End Of a File before expected.
137	Truncated record. Data record is more than 256 bytes long, and has been truncated at that point.
138	Device timeout. A specified I/O device does not respond after a period of repeated trying.
139	Device NAK A specified serial port or disk does Not Acknowledge attempts to use it.
140	Bus framing error. Serial-bus data is improperly formatted or inconsistent.
141	Illegal cursor position. A cursor function is generating an illegal line/row position for the current screen mode.
142	Serial frame overrun. Inconsistent serial-bus data.

Table G-1—cont. ATARI Error Codes and Their Meaning

Number	Meaning
143	Serial frame checksum error. Inconsistent serial-bus data.
144	Disk error. Disk system is not responding properly; the disk might be write-protected, its directory garbled or the drive is malfunctioning.
145	Read-after-Write error. Disk or screen handler finds a discrepancy between what was written to RAM and what actually appears there.
146	Operation not implemented. A specified operation cannot be carried out.
147	Out of graphics RAM. Insufficient amount of RAM for the graphics mode being implemented.
160	Invalid drive number. Program attempts to use an invalid disk drive designation.
161	Too many files. The program attempts to open more than three files at a time.
162	Disk full. All disk sectors are in use.
163	Device I/O error. The I/O system has encountered an error that it cannot handle.
164	File number mismatch. A discrepancy exists in the disk file management system.
165	Bad file name. Program cites an invalid file name.
166	POINT length error. A POINT statement refers to a sector byte that does not exist.
167	File locked. The program attempts to modify or erase a locked disk file.
168	Invalid XIO command. An XIO command is invalid, inappropriate or contains a syntax error.

Table G-1—cont. ATARI Error Codes and Their Meaning

Number	Meaning
169	Directory full. A given directory cannot contain more than 64 file names.
170	File not found. The program specifies a file name that is not included in the current disk directory.
171	Invalid POINT. A POINT statement refers to a disk sector that is not included in the current file.

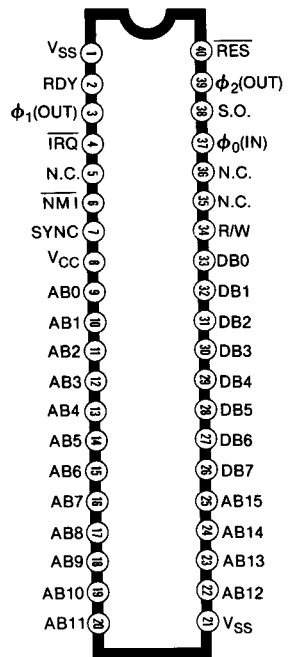
**Table G-2. ATARI IOCB (I/O Control Block)
Status Codes and Their Meaning**

HEX	DEC	Meaning
\$01	001	Operation complete (no errors)
\$03	003	End of file (EOF)
\$80	128	BREAK key abort
\$81	129	IOCB already in use (OPEN)
\$82	130	Nonexistent device
\$83	131	Opened for write only
\$84	132	Invalid command
\$85	133	Device or file not open
\$86	134	Invalid IOCB number (Y register only)
\$87	135	Opened for read only
\$88	136	End of file (EOF) encountered
\$89	137	Truncated record
\$8A	138	Device timeout (doesn't respond)
\$8B	139	Device NAK
\$8C	140	Serial bus input framing error
\$8D	141	Cursor out of range
\$8E	142	Serial bus data frame overrun error
\$8F	143	Serial bus data frame checksum error
\$90	144	Device-done error
\$91	145	Bad screen mode
\$92	146	Function not supported by handler
\$93	147	Insufficient memory for screen mode
\$A0	160	Disk drive number error
\$A1	161	Too many open disk files
\$A2	162	Disk full
\$A3	163	Fatal disk I/O error
\$A4	164	Internal file number mismatch
\$A5	165	Filename error
\$A6	166	Point data length error
\$A7	167	File locked
\$A8	168	Command invalid for disk
\$A9	169	Directory full (64 files)
\$AA	170	File not found
\$AB	171	Point invalid

Appendix H

ATARI 400/800 Hardware Details

Fig. H-1. 6502 microprocessor
pinout diagram.



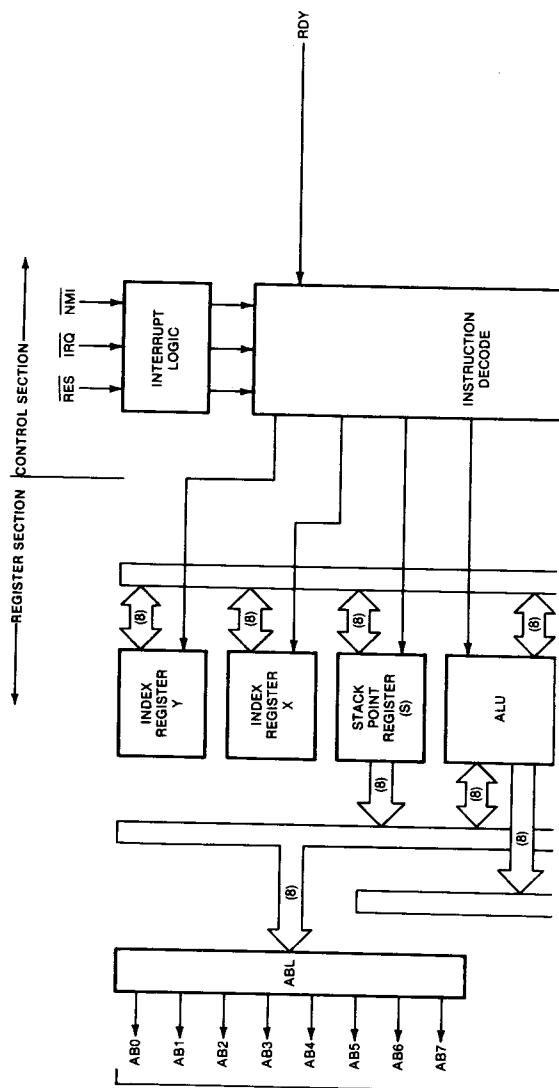
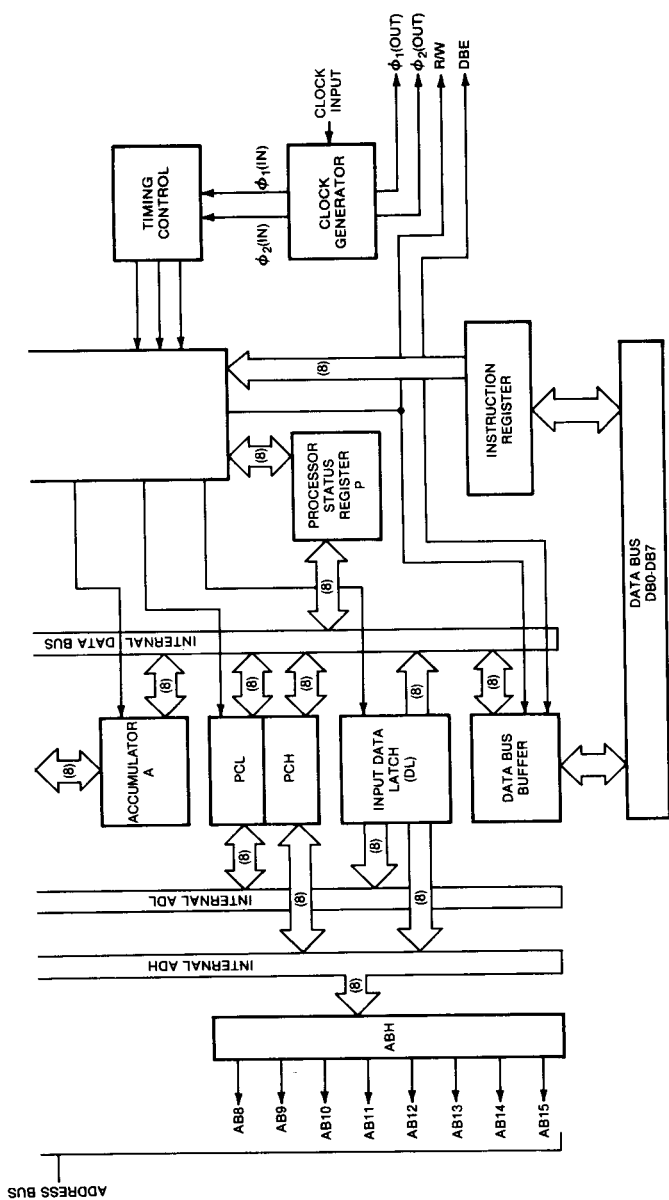
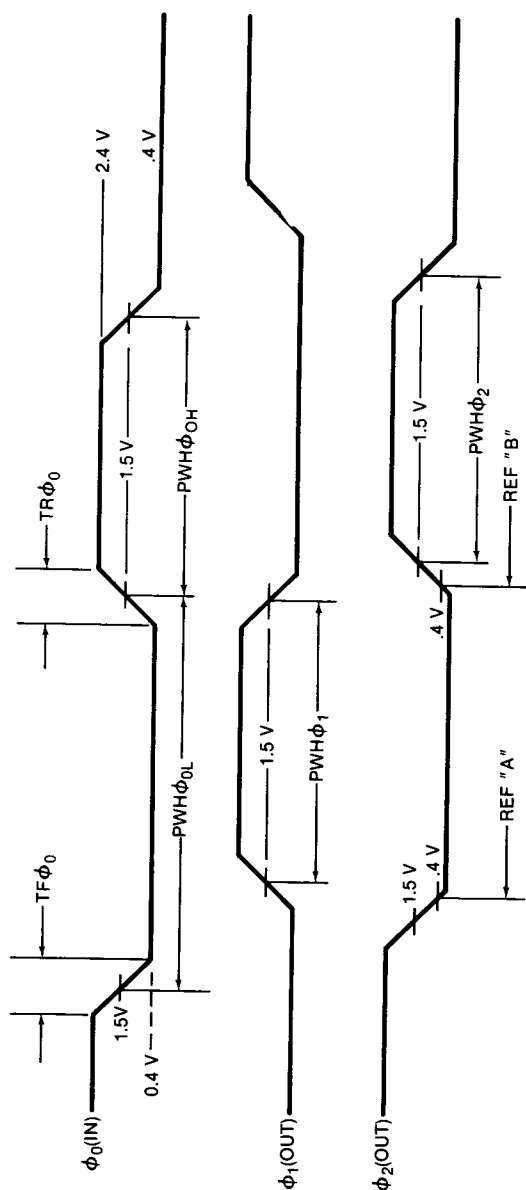


Fig. H-2. 6502 microprocessor



internal block diagram.



LEGEND

$\text{PWH}\phi_0 = \phi_0(\text{IN})$ PULSE WIDTH (MEASURED AT 1.5 V)
 $\text{TR}\phi_0, \text{TF}\phi_0 = \phi_0(\text{IN})$ RISE, FALL TIME
 $\text{PWH}\phi_1 = \phi_1(\text{OUT})$ PULSE WIDTH (MEASURED AT 1.5 V)
 $\text{PWH}\phi_2 = \phi_2(\text{OUT})$ PULSE WIDTH (MEASURED AT 1.5 V)

Fig. H-3. 6502 microprocessor clock timing diagram.

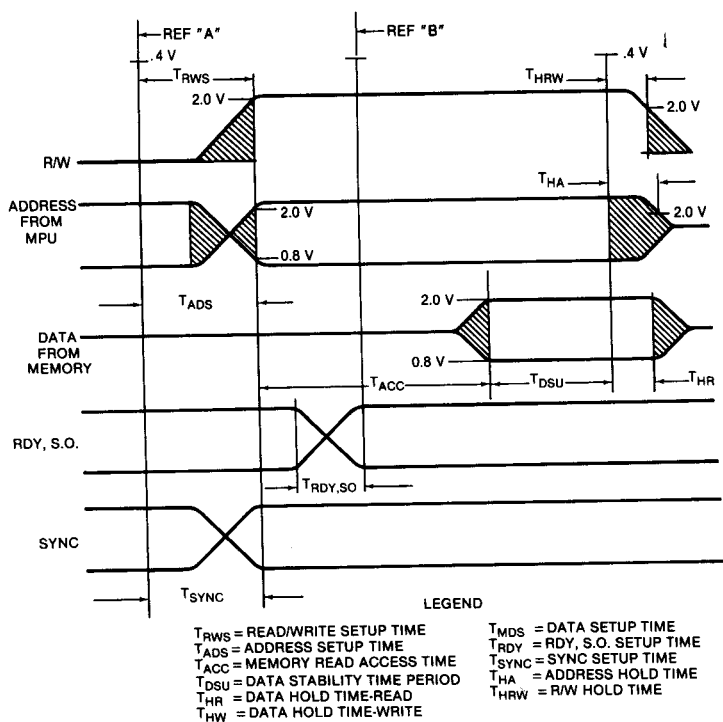


Fig. H-4. 6502 microprocessor READ-cycle timing diagram.

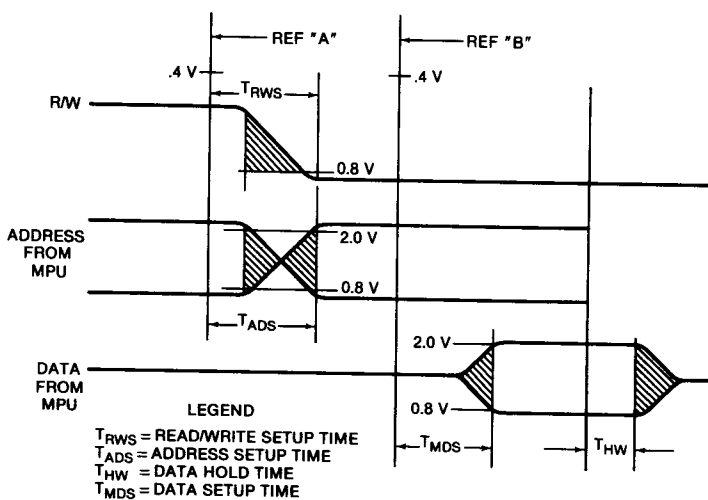
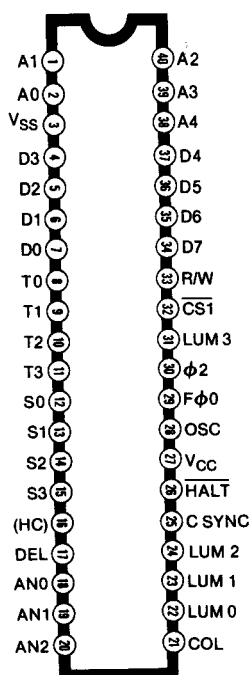


Fig. H-5. 6502 microprocessor WRITE-cycle timing diagram.

Fig. H-6. ATARI Home Computer GTIA (or CTIA) pinout.



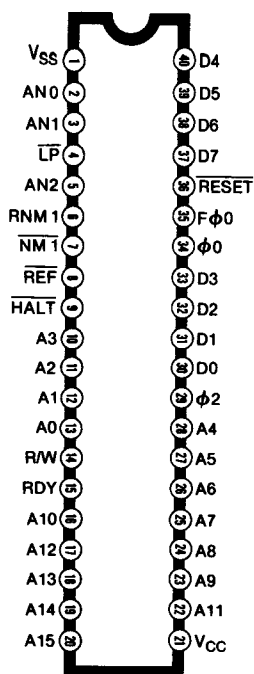
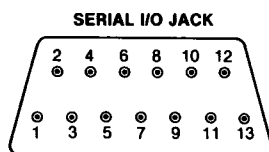


Fig. H-7. ATARI ANTIC pinout.



- | | |
|-----------------|------------------|
| 1. CLOCK INPUT | 8. MOTOR CONTROL |
| 2. CLOCK OUTPUT | 9. PROCEED |
| 3. DATA INPUT | 10. + 5/READY |
| 4. GROUND | 11. AUDIO INPUT |
| 5. DATA OUTPUT | 12. + 12 VOLTS |
| 6. GROUND | 13. INTERRUPT |
| 7. COMMAND | |

Fig. H-8. ATARI serial I/O jack pinout.

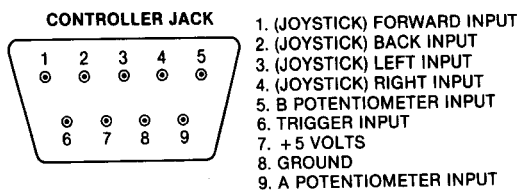


Fig. H-9. ATARI controller jack pinout.

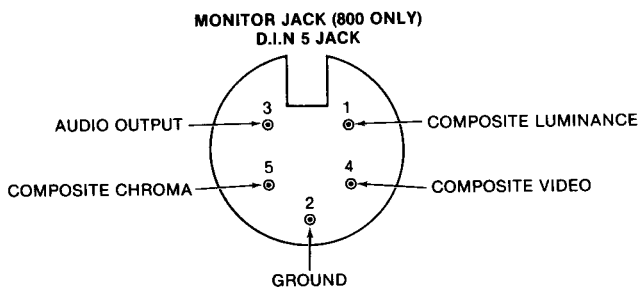


Fig. H-10. ATARI 800 monitor jack pinout.

Index

A

Accessing character set, 154-155

Addr, 152

Addresses

mode-0 screen, 436

mode-1 screen, 437

mode-2 screen, 439

mode-3 screen, 440

mode-4 screen, 442

mode-5 screen, 446

mode-6 screen, 217, 450

mode-7 screen, 455

mode-8 screen, 232, 460

mode-17 screen, 438

mode-18 screen, 439

mode-19 screen, 203, 441

mode-20 screen, 218, 444

mode-21 screen, 204, 448

mode-22 screen, 219, 452

mode-23 screen, 205, 457

mode-24 screen, 236, 464

starting and ending, 174

Animation, 244

Antenna switch, 16

ANTIC, 255

device, 345

instruction set, 300-303

map detail, 345

screen modes, 301

Append, 31

Arithmetic operators, 55

Arrays, DIM, 48-54

ASCII, 60

codes, 61

Asterisk option, 31

ATARI

ATASCII character set, 402

BASIC, 40

commands, 73-113

reserved words, 396

error codes, 472

internal character set, 135, 417

status codes, 477

system configuration, 15-18

ATASCII, 60

character set, 402

coded BASIC programs, 264-266

character set, 120

B

BASIC

commands, 68

ROM area: 40960-49151, 338

system RAM, 337

tokenized, 307-310

variables, 44-48

Baud rate, 285-286

Binary-to

-decimal conversion, 387-388

-hexadecimal conversion, 388-390

Bit

map, player/missile, 245-246

protecting player/missile,

249-250

maps for missile figures, 243-244

for player figures, 241-243

Boundary values for RAM addresses,

248

BREAK, 22

Buffer, INPUT string, 43

- Byte
 - least significant, 386
 - most significant, 386
- C**
- Cable assembly, 17
- CAPS/LOWR, 20
- Channel switch, 16
- Character
 - set, accessing, 154-155
 - ATARI ATASCII, 402
 - internal, 417
- CLOAD, 24
- CLOSE, 25
- Closing IOCB channels, 278-283
- CLR/SET/TAB, 22
- Codes, color register, 161
- COLCRS, 157
- Collision detection, 257-261
- COLOR, 148
 - registers, 183-184
 - codes, 161
 - screen mode 4, 204
 - mode 6, 204
 - sequences, 170
- Column/row
 - format, 145-147
 - screen formats, 184-188, 203
 - techniques, 148-149
- Command(s), 68-73, 308
- Common multiple-peripheral
 - system, 18
- Configurations, player and missile, 240-250
- Constants
 - numeric, 39-44
 - string, 39-44
- Control
 - keys, 20-22
 - statements, 68
- Conversion
 - binary to decimal, 387-388
 - to hexadecimal, 388-390
 - decimal to binary, 390
 - to hexadecimal, 384-385
 - hexadecimal to binary, 390
 - to decimal, 382-383
 - table, decimal, 391
- Copying
 - DOS files, 32-33
 - files to same disk, 28-30
- CSAVE, 24
- CTIA device, 341
- CTRL, 21
 - key operations, 428
- Cursor
 - position, 23
 - registers, 143
 - related BASIC statements, 143
- Custom
 - character
 - set for mode 0, 179-182
 - for mode 1, 179-182
 - for mode 2, 179-182
- D**
- DATA pointer, 50
- Decimal
 - conversion table, 391
 - to binary conversion, 390
 - hexadecimal conversions, 384-385
 - 2-byte decimal format, 385-386
- DELETE/BACK S, 21
- Deleting files, 30-31
- DIMensioning
 - numeric arrays, 48-54
 - string variables, 48-54
- Directory, 26-28
- Disk
 - directory, 26-28
 - drive I/O, 270-275
 - loading programs, 36-37
 - one, 17-18
 - saving programs, 36-37
 - duplicating, 33-35
 - formatting, 33
 - operations, routine, 26-37
 - utility operations, 26
- Display
 - list, 303-306
 - locating, 306-307
- DMACTL, 251
- DOS
 - files, copying, 32-33
 - menu, 26
 - system, 18
 - RAM usage, 338
- DRAWTO, 148
- Duplicating entire disk, 33-35

E

Editing features, screen, 22-23
ENTER "C:", 264-266
 "D:", 270-272
ESC, 22
Execute DOS, 26
Existing files, renaming, 31

F

Fast forward, 24
File(s)
 copying DOS, 32-33
 operation, 35-36
 deleting, 30-31
 designated, 37
 locking, 32
 name extension, 37
 nonprogram, 266-269
 renaming existing, 31
 unlocking, 32
Floating-point notation, 40
Formatting a disk, 33
Four-color modes 3, 5, 7, 182-195
Full-screen formats, 168-178
Function(s)
 summary, 68-73
 tokens, 310

G

Game-controller commands, 73
GET, 149-150, 268-269, 274-275
 in mode 1, 165-166
 in mode 2, 165-166
Global file names, 30
GPRIOR, 255
GRACLT, 251
Graphics
 commands, 73
 mode-0, 143
 mode-1, 152-178
 mode-2, 152-178
 operations, 184-188
 split screen, 144
GTIA device, 341

H

Hexadecimal
 to binary conversion, 390
 decimal conversions, 382-383
HITCLR, 257

Horizontal 6-position registers, 254
Hue, 115
 values, 118

I

INPUT, 266-268, 272-274
 string buffer, 43
Input/output commands, 68
INSERT, 21
Interface module, 18
Interference, 16
Internal character set, 417
Invalid numeric variable names, 47
Inverse-key operations, 429
I/O ROM area: 53248-55295, 340

J

Jack, peripheral, 77

K

Key codes, 426
Keyboard operations, special,
 19-22
Keystrokes, 426
Keys, control, 20-22

L

Least
 significant byte, 386
 nibble, 382
LIST "C:", 264-266
 "D:", 270-272
Loading
 binary programs, 277
 programs, 24-25
 with disk drive, 36-37
 under DOS, 275-277
LOCATE, 149-150
 in mode 1, 165-166
 in mode 2, 165-166
Locking files, 32
Logical operators, 62-63
LPRINT, 25
LSB, 386
Luminance, 115
 values, 118

M

Machine
 language options, 36
 routine, 300

Machine—cont.
 routine, 296-299
 Map development, 242
 Margins, mode-0, 147
 Math functions, 68
 Memory, 22
 MEM.SAV, 35
 Minimum working system, 15-16
 Missile figures, bit maps, 243-244
 Mode-0
 color registers, 144-145
 column/row format, 145-147
 graphics, 143
 margins, 147
 PRINT, 147
 screen, 143
 text screen, 144
 Mode-1
 color registers, 153-154
 graphics, 152-178
 Mode-2
 color registers, 153-154
 graphics, 152-178
 Mode-8 screen, 216-238
 Modes, screen, 115-143
 Most
 significant byte, 386
 nibble, 382
 MSB, 386
 Multiple-peripheral system, 18
 Musical scores, reproducing, 292

N

Names
 invalid numeric variable, 47
 valid numeric variable, 46
 variable, 44-48
 NEWROW, 160
 Nibble, 381
 least significant, 382
 most significant, 382
 Nonprogram files, 266-269
 Notation
 floating point, 40
 scientific, 40
 Null string, 44
 Numeric
 arrays, DIM, 48-54
 constants, 39-44
 variable(s), 44, 45-47
 names, valid, 46

Numerical variable names, invalid,
 47
 Numvar, 150

O

OLDROW, 160
 OLDROW, 160
 One disk drive, 17-18
 OPEN, 266
 Opening IOCB channels, 278-283
 Operating system ROM area:
 55296-65535, 346-348
 Operator(s), 55-66
 arithmetic, 55
 logical, 62-63
 order of precedence, 64-66
 relational, 56-62
 tokens, 309

P

Parity, 286-288
 PEEK, 150
 Peripheral jack, 17
 PIA
 device, 345
 map detail, 345
 Pixel, 193
 Play, 24
 Player
 and missile configurations,
 240-250
 figures, bit maps, 241-243
 Player/missile
 bit map, 245-246
 protecting, 249-250
 setting starting address,
 247-249
 color registers, 251
 colors, setting, 251
 figures, adjusting width, 250
 moving, 254-255
 width registers, 250
 graphics, initiating, 251-253
 terminating, 251-253
 Player/playfield priorities,
 255-256
 PLOT, 148
 POKE, 144
 operations for GRCTL register,
 252

POKEY

- device, 343
- map detail, 343

POSITION, 148

- statement, 159

Power switch, 16

Powers of 2, 389

PRINT, 41, 266-268, 272-274

Print-cursor features, 145

Program

- editing features, 22
- memory, 22
- recorder, 16-17, 23-25
 - connecting to system, 24
- I/O, 263-269

Programs, loading, 24-25

- saving, 24-25

Protecting player/missile bit map, 249-250

PUT, 149-150, 268-269, 274-275

- in mode 1, 165-166
- in mode 2, 165-166

R

RAM, 23, 150-152

- address, 312

READY, 16

Record, 24

Recorder

- connecting to system, 24
- I/O, working with program, 263-269
- program, 16-17, 23-25

Registers

- color, 183-184
- mode-0 color, 144-145
- mode-1, 153-154
- mode-2, 153-154

Relational operators, 56-62

Renaming existing files, 31

Reserved

- word list, 46
- words, ATARI BASIC, 396

RETURN, 20

Return to BASIC, 28

Reverse, 24

ROWCRS, 157

Running

- binary files under DOS, 275-277
- programs, 277

S

Saving

- binary data, 276-277
- programs, 276-277
- programs, 24-25
- with disk drives, 36-37
- under DOS, 275-277

Scientific notation, 40

Screen

- display lists, 300-307
- editing features, 22-23
- mode 8, 216-238
- modes, features, 115-143
- RAM address formats, 188-195
 - format for mode 4, 209-216
 - for mode 6, 209-216

SETCOLOR, 144

Setting starting address of player/missile bit map, 247-249

SHIFT, 20

Single-key operations, 426

6502 instruction set, 349

Sound

- commands, 73
- effects, experimenting, 292-293
- features, 289-293
- statement, 290-292

Split-screen graphics, 144

Stack RAM, 313

Starting and ending addresses, 174

- for mode-4 screen, 216
- for mode-6 screen, 217
- for mode-7 screen RAM, 198
- for mode-8 screen, 232
- for mode-19 screen, 203
- for mode-20 screen, 218
- for mode-21 screen, 204
- for mode-22 screen, 219
- for mode-23 screen, 205
- for mode-24 screen, 236

Start-up

- procedure, 17
- sequence, 18

Statements, summary, 68-73

Stop bits, 285-286

String

- buffer, INPUT, 43
- commands, 68
- constants, 39-44

String—cont.

 null, 44

 variables, 44, 47-48

 DIM, 48-54

Subscripted

 arrays, 51-54

 numeric variables, 51-54

Switch

 antenna, 16

 channel, 16

 power, 16

SYSTEM RESET, 28

T

Tape counter, 23

Text window, 144

Tokenized BASIC, 307-310

Translation modes, 286-288

Turn-on procedure, 18

Two

 byte decimal to conventional

 decimal format, 387

 color mode 4, 195-216

 mode 6, 195-216

TXTCOL, 157

TXTRW, 157

U

Unlocking files, 32

USR function, 293-299

V

Valid numeric variable names, 46

Variable(s)

 BASIC, 44-48

 DIM, 48-54

 names, 44-48

 invalid numeric, 47

 valid numeric, 46

 numeric, 44, 45-47

 string, 44, 47-48

W

Wild-card file names, 30

Words

 ATARI BASIC reserved, 396

 list, reserved, 46

 size, 285-286

Working system, minimum, 15-16

X

XIO command, 283-288

Z

Zero page, 313

Computer Direct

312/382-5050

Programmer's Reference Guide for the ATARI[®] 400[™]/800[™] Computers

Programmer's Reference Guide for the ATARI 400/800 Computers

- Introduces some of the fundamental concepts of BASIC language
- Provides a ready reference for the ATARI BASIC commands
- Discusses the 8 different screen modes and their 21 variations
- Features the enhanced animation package (player/missile/graphics)
- Covers the flexible I/O system
- Describes the memory map
- Contains the 6502 instruction set
- Covers the numbering system
 - Hexadecimal-to-decimal conversion
 - Decimal-to-hexadecimal conversion
 - Binary-to-decimal conversion
- Lists ATARI BASIC reserved words and tokens
- Gives ATARI character codes
- Features the ATARI keyboard codes
- Lists the error code numbers and their general interpretations

The information contained in this book is organized so the user can efficiently locate facts and application notes

Computer Direct

We Love Our Customers

Box 1001, Barrington, IL 60010