MASTER MEMORY MAP FOR THE ATARI

CRAIG PATCHETT
and
ROBIN SHERER

A Reston Computer Group Book
Reston Publishing Company Inc.
A Prentice-Hall Company
Reston, Virginia

Library of Congress Cataloging in Publication Data

Patchett, Craig.
Master memory map for the Atari.

"A Reston Computer Group Book."
l. Atari computer-Programming. 2. Memory maps
(Computer science) l. Sherer, Robin. 11. Title.
QA76.8.A821,38 1984 001.6412 84-476
ISBN 0-8359-4242-2

Edited by Graham Patchett and Sylvia I. Smith
Illustrated by Frank Hill

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

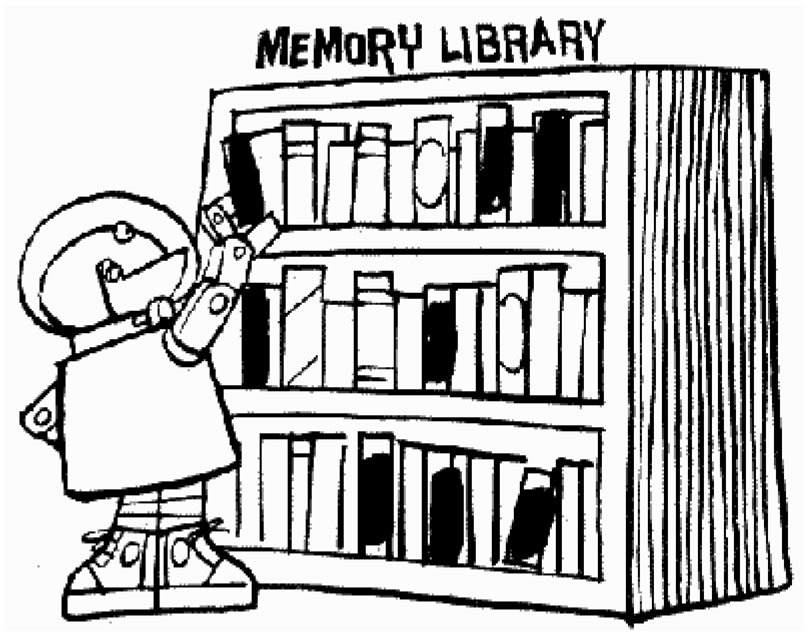
Contents

APPENDICES

PREFACE

WHAT HAVE I PURCHASED?

You are the proud owner of a detailed collection of the internal workings of your Atari computer. By simply using the POKE and PEEK commands from BASIC, which we will explain in a few pages, you can change the numbers within many locations in memory. This book is a “map” to find where you are in the Atari’s huge address space of 65536 memory locations. It is perfect for both beginners and experts. For the new computer owner, we start with simple explanations of computer terminology. Next come many examples of the various “tricks” we will show you. The more advanced programmer will find that the book is absolutely necessary as a reference to the large number of things you can do with the Atari. Few people can memorize over 1000 locations in a computer, and then recall what the individual bits in each location do.



If you are a beginner, start with the glossary to become familiar with some of the “computereze” necessary to understand the machine. Next, study the first few sections to learn about the POKE and PEEK commands and hexadecimal-to-decimal conversion. Now begin to read through the book starting with memory location 0. For many locations we offer a complete and lengthy explanation in an appendix, so you’ll want to make sure you also check out the appendices.

Don’t be afraid to put a number into the computer’s memory. All you can hurt is your pride. The computer will just “go to sleep” if you tell it to do something impossible. If that happens, then all you have to do is turn it off and on again and continue to explore. We will provide you with examples to illustrate techniques and ideas. Try changing the numbers in each example to “see what happens if…” You learn best if you type in the examples yourself, but because there are so many, and we KNOW how valuable your time is, we have one more offer for you:

SEND US MONEY!

If you don't want to tire your fingers, send $9.95 to cover the traditional postage, handling, and media costs to

EDUCATIONAL SOFTWARE, inc
~~4565 Cherryvale Ave.~~
~~Soquel, CA 95073~~

We will send you all of the programs in this book on your choice of tape or disk.

A BONUS!

If you discover a new, unpublished use for a location in the computer, send it to us. In return, we will send you a free program.

SOURCES

The following sources were instrumental in understanding the purposes of some of the more esoteric locations:

De Re Atari - Atari, Inc., ~~1312 Crossman Ave., Sunnyvale, CA 94086~~

DOS Listing - Atari, Inc., ~~1312 Crossman Ave., Sunnyvale, CA 94086~~

Inside Atari DOS - Compute! Books

Hardware Manual - Atari Inc., ~~1312 Crossman Ave., Sunnyvale, CA 94086~~

Mapping the Atari - Compute! Books

OS Listing - Atari Inc., ~~1312 Crossman Ave., Sunnyvale, CA 94086~~

GLOSSARY

Here at Educational Software we get tired of computer terminology. However, many of these words are becoming a part of our language. As we explain the inner workings of the Atari, we will have to refer to some of the following words. If your find some term we forgot to mention here, it's probably because it is fully explained in the appendices or at the memory location it pertains to. You should also read your BASIC manual in order to understand the terms that have to do with the BASIC language.

6502: This is the heart of the computer, the chip that bosses everybody around. Actually, a lot of people even refer to the 6502 as being the computer, since it does have almost all of the brains.

Accumulator: This is a location that is used to temporarily store the results of logic and arithmetic operations. The main accumulator is inside the 6502 chip, but sometimes memory locations are also used as an extra accumulator.

Address: The number assigned to an individual memory location. Each byte in the Atari has its own unique address, much like a house has a street address. The main use of this book is to provide you a roadmap to each address so you don't get lost.

Algorithm: A general procedure, plan, or method that represents how your program will be written.

ANTIC: This is a chip in the Atari computers that figures out what the screen is supposed to look like.

ASCII: The **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange (pronounced ASK-KEY). Everyone needs a standard or reference to refer to. This allows us all to speak to each other in the same terms. Humans use dictionaries to speak the same words. In the case of computers, ASCII allows one computer to understand the letters and numbers created on another computer. Atari computers do not follow a true ASCII, but have their own code instead which we explain later.

Assembly language: This is a programming language, just like BASIC, except it talks the computer's language instead of having to go through a translator. See **machine language** as well.

ATASCII: **ATA**ri **S**tandard **C**ode for **I**nformation **I**nterchange. This is the code the Atari uses to convert letters to numbers and vice versa. See your BASIC manual to find out how it differs from ASCII.

Baud: The rate of transmission of information conveyed between two computers. You usually say “Baud Rate” meaning how fast the two computers are talking to each other. This rate is determined by the bits per second that are being transferred. You encounter this term if you are using a modem, printer, disk drive, terminal or other device that needs to talk to a computer to work. Typical speeds of information transfer are 300, 1200, 2400, 9600, and 19200 bits per second.

Bit: The smallest piece of information the computer can handle. There are eight bits in a byte. Each bit can either be “on” or “off.” See the section on Bits and Bytes for a complete description. Sometimes in this book you’ll see “-” for the value of a bit. This just means that it doesn’t matter if that particular bit is on or off.

Bit mapping: This refers to the process of turning individual bits on and off without changing the rest of the byte.

Boot: No, this isn’t even close to what it sounds like. “Booting” a program means loading it in when the computer is turned on. For example, if you hold down the START button while turning on the computer, the computer will beep. This means that it expects a boot cassette to be in the cassette player. When you turn on the computer with the disk drive on, you will boot DOS. In other words, any program that loads in without you having to tell it to load is a boot program.

Boundary: As in “4K boundary.” This is the end of a block of memory. For example, a 1K boundary would be the end of 1024 byte block.

Buffer: A storage place, usually temporary, where information can come and go without disturbing things.

Bus: A bus is a system of electrical lines shared by all devices that are connected to it. This is a convenient way for these devices to share data. It works just like a party-line telephone. Different parts of the computer talk to each other by getting on the bus and sending messages.

Byte: Pronounced BITE. A collection of eight bits. Each memory address consists of one byte. Since we know at this point bytes and bits can be confusing, we provide a special section elsewhere in the book, called BITS and BYTES, to explain it to you.

Checksum: A checksum is a special byte that the computer uses after talking to something to make sure it understood what was said correctly.

CIO: Central **I**nput/ **O**utput. This is Atari’s main I/O routine.

Cold Start: A routine the computer goes through after you turn it on and before it lets you tell it what to do.

Color clock: A unit of measurement for the screen. A color clock is the width of a pixel in graphics mode seven. That means that the screen is 160 color clocks wide from border to border.

Controller jack: What you plug your joystick into.

CTIA: This chip takes care of translating the data coming from ANTIC into something the television set can understand.

Cursor: The position on the screen where the next character or pixel will appear. In graphics mode zero, you can see the cursor; it's the white box.

Data: Any kind of information that is needed by a program or by the computer.

Default: When you first turn on the computer, each memory location will contain a value. These initial values are called defaults, meaning that this is what these locations will equal if you don't change them.

Device: Anything that the computer has to talk to is called a device. This includes the disk drive, printer, and even the keyboard and television set.

Disable: To turn off. By disabling the BREAK key, for example, you can prevent someone from accidentally stopping your program.

Display list: The program for ANTIC that describes what the screen is to look like.

DLI: **D**isplay **L**ist **I**nterrupt. See **interrupt**.

DMA: **D**irect **M**emory **A**ccess. The process of getting data from memory to put on the screen.

DOS: **D**isk **O**perating **S**ystem. A program that controls the use of the disk drive. See **OS** as well.

DUP: **D**isk **U**tilities **P**ackage. This is a bunch of routines to do various things on the disk drive. The DOS menu is actually a list of these routines.

Enable: To turn on. The opposite of disable.

File: A whole bunch of data stored on disk or cassette.

Flag: A signal that a certain condition has been met. In many BASIC programs, variables are used as flags, as demonstrated in the following example:

```
10 IF A=B AND C=D THEN FLAG =1
      .
      .
      .
50 IF FLAG= 1 THEN 100
```

Floating point: A type of arithmetic where the decimal point can appear anywhere in the numbers (i.e., it can float around). An example of such numbers would be 1.0, 23.97, and 1.45678E+04. Floating point numbers take up much more memory than fixed point (integer) numbers.

FMS: File **M**anager **S**ystem. This is a group of routines, or handler, to help the computer talk to the disk drive.

GTIA: A fancy version of CTIA.

Handler: A series of routines that tell the OS how to handle a particular device.

HBLANK: Horizontal **BLANK**. The television set draws the screen one line at a time, from top to bottom and left to right. HBLANK is the time during which it is moving from the end of one line to the beginning of the next.

Hi-res: Pronounced "high rez." This is an abbreviation for "high resolution," which refers to a graphics display with very small dots.

Immediate mode: Using the computer without running a program. For example, if you type in

```
PRINT 3 + 2
```

and then press RETURN, you will get a result of 5 on the screen **immediately**.

Index: This is a variable used to keep track of where we are in a loop. For example, in the following statement X would be an index:

```
FOR X=1 TO 100
```

Internal: If something is internal, then that usually means it is built into the computer.

Interrupt: An interrupt is something that interrupts whatever the computer is doing and tells it to do something else before it continues. You should also see **DLI**, **IRQ**, **NMI**, and **VBLANK**.

I/O: **I**nput/**O**utput (I/O) is nothing more than a fancy way of referring to the computer talking to a device, or vice versa.

IOCB: **I**nput/**O**utput **C**ontrol **B**lock. This is a place that you use to talk to CIO. See the appendix on I/O (appendix 7).

IRQ: **I**nterrupt **R**e**Q**uest. This is a kind of interrupt that you can tell the computer to ignore (the 6502 can enable or disable it).

Jiffy: A jiffy is one sixtieth of a second, the time that it takes the television set to completely draw the screen once. In European (PAL) systems, a jiffy is one fiftieth of a second.

Jump: The same thing as GOTO. The expression "jump through location" means that the computer will GOTO the address stored in that location.

K: As in 1K, 8K, 16K, etc. I K is equal to 1024 bytes.

Logical line: A logical line is the space that a program line takes up. It can be one, two, or three screen lines (try typing in a BASIC line that is more than three screen lines).

Machine language: Machine language is a way of talking directly to the 6502 chip. Other languages like BASIC have to be translated into machine language before the 6502 can understand them. That takes time, which is why machine language programs run so much faster than BASIC ones. In case you're wondering what the difference is between machine language and assembly language, not much. Machine language is just a bunch of numbers. Assembly language gives these numbers names so that they make more sense.

Masking: When you're bit mapping, you have to have a way of ignoring the bits you're not interested in. This process is called masking, since you essentially place a mask over the bits you don't want to look at.

Nibble: This is going to sound funny, but I swear it's the truth. A nibble is half a byte, or four bits.

NMI: **N**on-**M**askable **I**nterrupt. Unlike IRQs, you can't tell the 6502 to ignore this kind of interrupt. DLIs and V BLANK interrupts are both NMIs.

NTSC: A name for the television system that is used in North America. European television is slightly different and uses a system called PAL.

Offset: If you have a whole bunch of bytes making up a table of values or a buffer or something similar, then the offset is the number of the byte in this bunch that you are currently interested in.

OS: Operating **S**ystem. Its job is to make the computer run. You can think of the OS as the coach directing the players in a game. We can change some of the numbers in the operating system to make the computer do what we want, instead of what it normally does.

Page: Computer memory in the Atari is divided into 256 sections, called pages. Each page consists of 256 bytes. The pages are numbered zero through 255, and you can tell what page a particular location is in by looking at the high byte of its address. For example, location $09AB would be in page nine. See the section called "Computer Mathematics" for an explanation of what a "high byte" and "low byte" are, and also for an explanation of "hexadecimal," which is what that funny number with a "$" in front of it is.

PAL: The television system used in Europe. See **NTSC** as well.

Parallel: There are two ways that the computer can talk to something else. One of these is called parallel I/O, which simply means that the data is sent out one byte at a time. See **serial** for the other.

PIA: This chip takes care of the controller jacks.

Pixel: A fancy word for a dot on the screen.

Playfield: Anything that appears on the screen other than a player or missile (see the appendices for a description of players and missiles).

Pointer: A pointer does exactly what it sounds like: points somewhere. Usually this "somewhere" is the location of some information that is needed. The pointer holds the address of this location.

POKEY: † value you want them to have.

ROM: Read **O**nly **M**emory. Computer memory that you can't change with the POKE command or anything else (it's OK to PEEK them though). ROM locations even remember their values after you turn the computer off! BASIC and the Atari operating system are stored in ROM.

Scan line: If you look really closely at the screen, you'll see that it's made up of a whole bunch of tiny horizontal lines. These are called scan lines and are the height of a graphics mode eight pixel.

Screen memory: A bunch of bytes somewhere in memory (usually at the end) that ANTIC converts into a picture and sends to GTIA or CTIA which puts it on the screen. In other words, this is where the data that is to appear on the screen is stored. In case you don't understand the difference between this and the display list, the display list tells ANTIC how to interpret the screen memory (i.e. where is it, does the data represent characters or pixels, how big are they, etc.).

Sector: A group of 128 bytes on the disk. It may be difficult to do, but try to imagine a disk being made up of 40 concentric rings. Now imagine cutting the whole disk into 18 equal-size wedges. Each of these wedges will have 40 pieces for a total of 720 pieces altogether. Well, each of these pieces is a sector.

Serial: This is a method of I/O that sends data out one bit at a time rather than one byte at a time. See **parallel** also.

Shadow register: A shadow register is a RAM location that acts like a messenger for a chip location. Any changes to the shadow register are sent to the chip, and vice versa. This is necessary because a chip location can't be changed permanently, and so it relies on its shadow register to get information from you.

SIO: Serial **I**nput/ **O**utput. This refers to a routine in the OS that takes care of serial I/O. See **serial**.

Timeout: Sometimes a device needs a little time to think and breathe, so it takes a timeout. If it takes too long a timeout, however, the computer gets upset and refuses to talk to it anymore.

User: You, anybody, or anything that uses the computer. BASIC is considered a user by the OS, and the OS is considered a user by the 6502. Similarly, BASIC considers your program a user, and your program considers anyone that runs it a user.

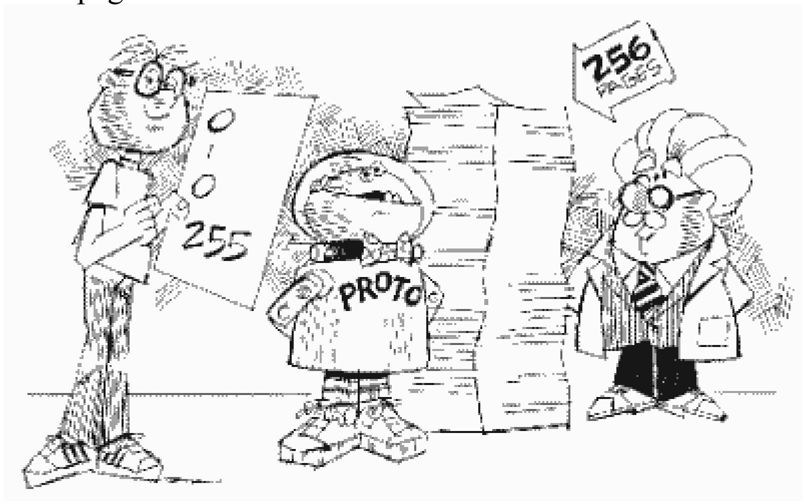
VBLANK: Vertical **BLANK**. We already saw that the television set draws the screen over and over from top to bottom and left to right (see **HBLANK**). VBLANK is the time during which the television set is going from the bottom of the finished screen back to the top to start drawing again.

Vector: This is another kind of pointer. It refers to the starting address of a routine. The computer needs to know where to look for things, and the vectors help it along the way. Usually a vector references the starting address of a machine language subroutine.

Warm Start: A routine the computer goes through after you press SYSTEM RESET and before it lets you tell it what to do.

WHAT IS A MEMORY LOCATION?

Good question! Your Atari has many “places” within it that can contain numbers. These places we call memory cells, locations, or addresses. You may use any of these words to mean the same thing, but “address” is the more formal term that you will hear computer programmers using. Since the computer can remember the numbers in each of these places, it is common to call them the computer’s MEMORY. Memory is like many blank pages of paper. Each page can hold only 256 numbers, and we can have up to 256 of these pages.

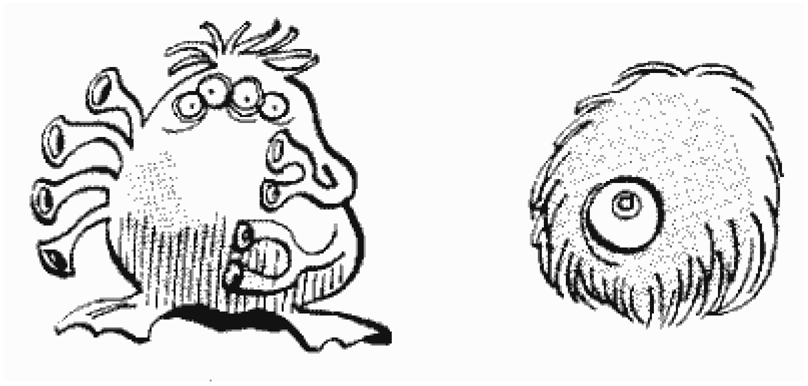


So where does this leave us? Well, 256 locations per page times 256 pages gives us 65536 locations total that CAN be in your computer. Computers count in terms of something called a “K.” For reasons beyond my control, one K of memory is actually 1024 locations. Why? I’ll explain all in a few pages. For now divide 65536 by 1024 to get a possible memory size for the Atari of 64K. Oops! When you bought your Atari, the salesperson probably told you it had only 16K or 48K. Let’s solve this very great mystery.

The Atari is capable of using 64K of memory. The factory gives you 16K ROM and 16K or 48K RAM when you buy the computer (we’ll explain ROM and RAM later), but there is a catch (of course). A lot of the memory that comes with the Atari is already filled up with numbers that tell the computer how to run. The blank area available for you to use is a lot less than you’re led to expect. If you want more memory, you have to go down to your friendly dealer and buy it. Memory additions come in 16K, 32K, and 48K plug-in “cartridges.” My grandmother once told me, “Do yourself a favor, buy the large economy size, you’ll need it.” She was talking about soap, but her wisdom applies to computers as well.

Now that you understand how much memory you have, let’s talk about it a little. Each memory address can hold numbers from 0 to 255. Computers start counting at 0 because “nothing” is a very valid piece of information. I certainly worry when my wallet has “0” in it! Let’s learn how computers count.

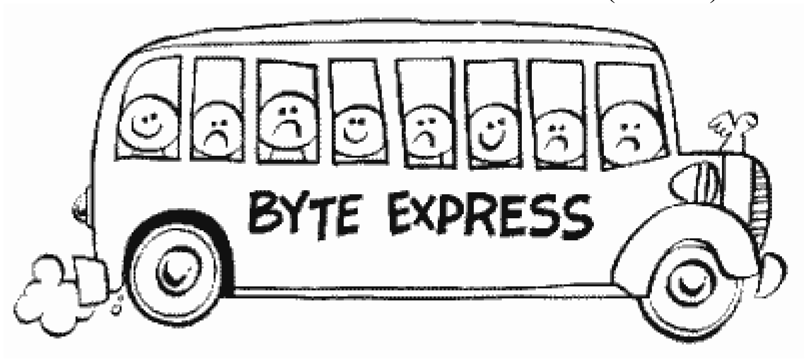
BITS AND BYTES



A BYTE is really not complicated at all. It is simply a group of eight BITS. When eight BITS are structured into a BYTE, then each of those BITS has special significance. You look puzzled! What, you say, is a BIT?

A BIT is the smallest piece of information that a computer can deal with. To help understand how BITS are used by the computer, it may help to imagine the microprocessor as a bus station. This bus station is on a single-lane road. That means a bus can only travel in one direction at a time as there is not enough room for two buses to pass each other. Therefore, a bus may either be arriving at the station or departing. The microprocessor, or bus station, can schedule its bus with a signal light that says “I am accepting arrivals” or “I am sending departures”.

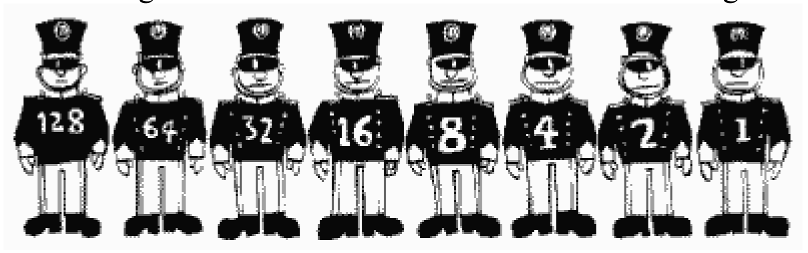
In fact, in real computer hardware architecture, the wires that carry information to and from a microprocessor are called the DATA BUS. We don’t need eight separate INPUT and eight separate OUTPUT wires because, like the single lane road connected to the bus station, the wires are bi-directional. In other words, information can either be arriving (INPUT) or departing (OUTPUT), but not both. The microprocessor also has a signal of its own that determines whether it will receive (INPUT) or send (OUTPUT) information.



Let’s take a closer look at that bus. It is known as the BYTE express, has eight seats, and always carries eight passengers. Those passengers are little messengers known as BITs, and, as a group, they are known as a BYTE. These messengers, or BITs, are rather moody. They are either turned "ON" or they are turned "OFF." That is called BINARY as they are BI-STATE signals, ON being a "1" state and OFF being a "0" state. Their vocabulary is just as limited… the only thing they are willing to tell you is their mood. Now how do we get any meaningful information out of a group of eight little messengers standing in front of us, each screaming "ON" or "OFF" at one time?

Well, when the bus arrives, we could have the whole BYTE stand in front of us and then count everyone who is turned "ON." That would give us the capability of counting to eight. Seems pretty limited, doesn't it? Hmmm, the group really needs a leader. That leader will be the first BIT on the left. We'll call that BIT the **M**ost **S**ignificant **B**it, or the **MSB**. The last BIT on the right will be the **L**east **S**ignificant **B**it, or **LSB**. Terrific! Now that we have a group leader and a group follower, all the BITs should be given a rank.

Handing out ranks is serious business and much thought should be given to it. We can start with the **LSB** and assign that BIT the rank of "1," since it is the Least Significant Bit. We can be easy on everyone if we just double that rank for the next BIT in line. So, why not just keep doubling the rank for the next BIT in line and so on until we get to the **MSB** or Most Significant Bit. Now our BYTE looks something like this:



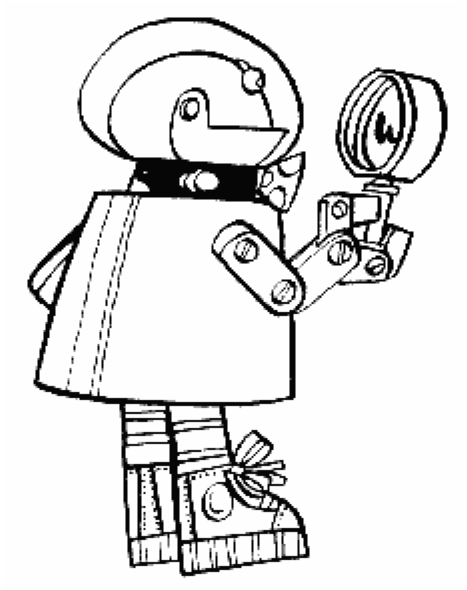
What have we gained? More than meets the eye! When the BYTE gets off the BYTE express and each BIT starts telling us what its current mood is, we can make a different and more meaningful interpretation out of the little guys. If everyone is turned "OFF" except the fourth BIT from the right, for example, we can check the rank of that BIT and find it is eight (8). Unknown to the BITs, they have brought us a message and the message is "8".

Aha, what if we want the bits to get on the bus and carry a message that says "9". This is a problem because there is no bit with a rank (value) of 9. What to do? I guess the next best thing is to be very nice to the bits that have values of one and eight and turn them both on. When we look at the BYTE now we see 1 + 8 which gives us our 9. Easy. Even my student Nerdwell can count that high. In fact, Nerdwell can count to 255 because if you add up all the values of the bits, you get 255:

1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255

HOW TO PEEK

In using this book, we are going to change the values that are stored in many of the BYTEs that make up memory. To make the discussion easier, we will usually talk about the bytes as locations or addresses. These addresses are just like those on mailboxes on a long street. They start at 0 at one end of the street and increase until they reach 65535 at the other end (64K… remember). Yes, a computer's memory represents a long street. Let's look now at one such address. To see what is inside the location, we will use a BASIC command called PEEK. Prototype loves to peek at things.



To PEEK at memory location 764, type in this line:

```
10 PRINT PEEK (764):GOTO 10
```

Type "RUN", press the RETURN key, and watch the number 255 print over and over on your screen. Now press the space bar and you should see 33 printing. What is happening? 764 is the location that tells you what key is being pressed. 255 means no key and 33 means the space bar. Now you can see how a program could know if you have pressed a certain key. Try a program something like this:

```
10 A = PEEK (764)
20 IF A=33 THEN PRINT"YOU PRESSED THE SPACE BAR"
30 GOTO 10
```

Now you can see why it is important to read this book. It would be very difficult to know which location to PEEK at without a map of your computer's memory. It's a good thing that old Professor Von Chip is such a good mapmaker.

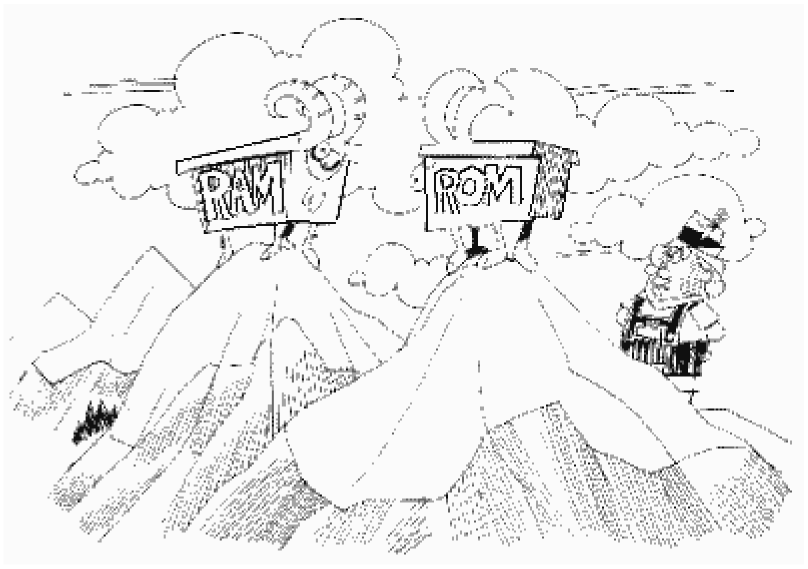
HOW TO POKE

Now that we can look at memory locations, we also want to be able to change what is inside them. Your wish is the professor's command. Later in the map you will learn that a value of one in location 752 will cause the cursor to disappear (the cursor is the little white box on the screen). This can come in handy when you have a whole bunch of text on the screen and you don't want the cursor up there with it. Since location 752 doesn't normally contain a value of one, we must change it. The POKE command in BASIC will do this for us:

```
POKE 752, 1
```

Wow! That was easy. Now you know that POKEing and PEEKing are what this book is all about. For the most part the book will just tell you different numbers to POKE or PEEK.

ROM AND RAM



We encounter another small problem when we mess with memory. Some memory addresses will allow us to read what is there, but will NOT allow us to write to them. Memory that we can both read from and write to is called RAM, which stands for **R**andom **A**ccess **M**emory. This means that we can put numbers into these kinds of memory locations as well as look at what they already contain. The other kind of memory is called ROM, **R**ead **O**nly **M**emory. It is just what its name implies - we can only read what number is inside a ROM location not change it.

Now the next logical question is how do you know which is which? We will simply tell you as we go through the locations.

COMPUTER MATHEMATICS

This section will open new vistas in your horizons. We are going to learn how the computer deals with numbers larger than 255, and also how to use the hexadecimal numbering system. Just think of it. Soon you will be a computer brain just like Prototype.

You must always use decimal numbers with a POKE statement. This means that sometimes you will have to convert between binary, hexadecimal and decimal. The "Bits and Bytes" section covered binary numbers where we turn on and off individual bits. In that section you saw that each memory location can only hold numbers up to 255. To store a value larger than that, we just use two locations in a row.

As an example, look at memory locations 88 and 89 which are called SAVMSC. These locations hold a number that tells where the top of the screen is. Because the screen is usually somewhere near the end of memory (it starts at location 40000 in a 48K Atari), at an address that is way beyond the 255 limit for one memory location, the computer needs two locations to store the address. Remember the ranking of the bits where the MSB, or bit seven as we called it, was valued at 128? If we double that rank again, we get 256. Here is the trick to computer math. Since there is no bit eight to give such a rank to, we give the rank to the entire next byte of 8 bits. Now we just see what the total of this second byte is and multiply it by 256. Figure 1 is a sample:

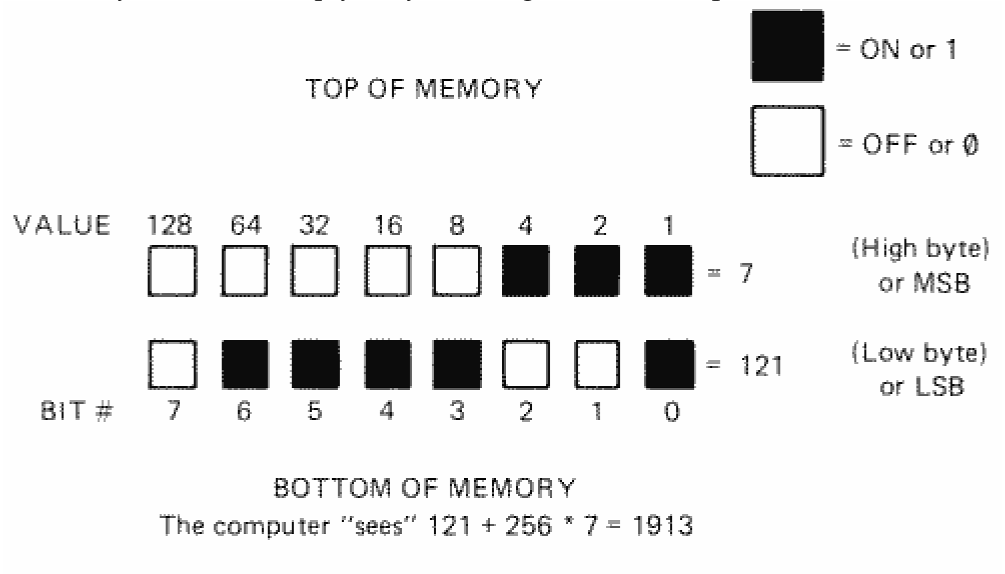


FIGURE 1. Bit map

When using two byte numbers, we call the first byte in memory the LOW BYTE because it stores the LOWER VALUE. It can only count from 0 to 255. The second number counts in multiples of 256 and is called the HIGH BYTE.

Let's try to PEEK and POKE a two-byte number. Suppose you wanted to fool the computer into thinking that the screen was in a different place in memory. Such a trick can come in handy when you want to have more than one screen at the same time. First let's look at memory and see where the computer thinks the screen is now. Here's a program to do this:

```
10 SCREEN=PEEK (88)+PEEK (89)*256
20 PRINT SCREEN
```

We know to look in locations 88 and 89 ROM the main part of this book that lists each location in numerical order. This two-byte location is called SAVMSC and points to where the computer thinks the first byte of screen memory is.

When you run the preceding program, the address that gets printed out will vary from 7232 to 40000, depending on how much memory your computer has. What we're going to do is add 480 to this address so that the computer will think that screen memory starts halfway down the current screen. All this means is that no text will print in the top half of the screen, and you'll be able to type below the bottom of the screen (although you won't be able to see anything there).

Let's go ahead and replace the old value of SAVMSC with a new number that is exactly 480 more than the old one. Add the following lines to the preceding ones:

```
30 SCREEN=SCREEN+480
40 SCRHI=INT (SCREEN/256)
```

Line 40 finds out what number goes into the high byte of SAVMSC. Remember that the high byte counts the number of pages, or multiples of 256, and that's why we divide by 256.

```
50 SCRLO=SCREEN-SCRHI*256
```

Now we multiply the new high byte by 256 and subtract it from the total to get the low byte.

Finally we place the new values in memory:

```
60 POKE 88,SCRLO:POKE 89,SCRHI
```

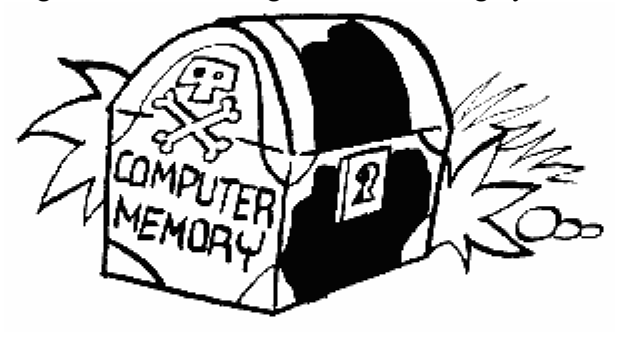
Let's take a look at what the values would be in each step of the program, assuming you have a 16K Atari:

```
10 SCREEN =64+60*256=15424
20 PRINT 15424
30 SCREEN =15424+480=15904
40 SCRHI=INT(15904/256=INT(62.125)=62
50 SCRLO=15904-62*256=15904-15872=32
60 POKE 88,32:POKE 89,62
```

The high byte can count from 0 to 255 just like the low byte. This means the largest number we can have using a two-byte address is:

$$255+(255*256) = 65535$$

If we count 0 as a number (since the computer does), that gives a total of 65536, which brings us back again to 64K. We have now come full circle in our discussion, so it is time to go on to something else to challenge you.



HEXADECIMAL NUMBERS

Come back. You don't have to run away at the sound of those words. Hex, as its friends call it, is not nearly as hard as everyone thinks it is. As a matter of fact, it's really quite simple. But, just in case you don't believe us, I provide both decimal and hexadecimal numbers throughout the book. Now, though, we're going to learn about hex together.

The main use for hex is in assembly language programming. We're not going to worry about that now though, because it's not really important. Instead, we're going to go back to grade school, where we first learned all about the number system. As you'll recall, we use the decimal number system, which means that everything is based on powers of 10. For example, the number 452 is equal to 4*100+5*10+2*1, right? In other words, 4*10^2+5*10^1+2*10^0. Now you're not going to believe this, but the only difference between decimal and hex is that hex is based on powers of 16 instead of 10. For example, 452 hex would equal 4*16^2+5*16^1+2*16^0 or 4*256+5*16+2, in other words, 1106. Pretty simple, huh? Actually, there's one more thing that I should probably mention. The number 9 in hex x is the same as 9 decimal, but 10 hex is 16 decimal (1 * 16^ 1). So how do you write 10, 11, 12, 13, 14, and 15 decimal in hex? Try A, B, C, D, E, and F! That's why hex numbers look so confusing. So, for example, F in hex equals 15 decimal, 1F equals 31 (16+15), and so on. Oh, and by the way, binary is actually the same as hex and decimal, except it counts in powers of 2!

Don't worry, I'm not going to leave you yet. A few more examples should make you a little more comfortable about hex and how to use it, but first a chart (Table 1) to help us out.

Table 1. Hex Conversion Chart

Column #	4th	3rd	2nd	1st	# HEX
	4096	256	16	1	1
	8192	512	32	2	2
	12288	768	48	3	3
	16384	1024	64	4	4
	20480	1280	80	5	5
	24576	1536	96	6	6
	28672	1792	112	7	7
	32768	2048	128	8	8
	36864	2304	144	9	9
	40960	2560	160	10	A
	45056	2816	176	11	B
	49152	3072	192	12	C
	53248	3328	208	13	D
	57344	3584	224	14	E
	61440	3840	240	15	F

DECIMAL TO HEX

Let’s take the number 9304. Look in the chart to find the largest number that is SMALLER than 9304. That would be 8192, which is the second number in the fourth column. This means the hex number will have four digits because you found the decimal value 8192 in the fourth column. Write down the “hex #” from the chart in the fourth place:

2---

Now subtract 8192 from the original number of 9304 to leave a remainder of 1112. Looking at the chart again we find the nearest SMALLER number of 1024 in the third column. Put down the corresponding hex # and subtract 1024 from 1112:

24--

Do the same for the new remainder of 88 to find the hex # digit for the second and first place of the hex number. Don’t forget to mark hex numbers with a “$” as is standard with computer types like us. Here is the complete example (Example 1):

Decimal-to-Hex

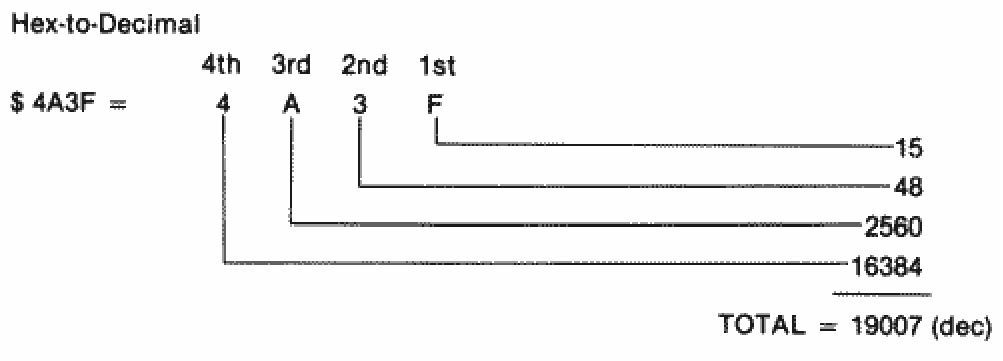
```
 9304
-8192 (largest # less than 9304 on chart, 4th column) ----→ 2
 1112
-1024 3rd column -------------------------------------------→ 4
   88
   80 2nd column -------------------------------------------→ 5
    8 1st column -------------------------------------------→ 8 = $2458
```

Example 1. Decimal-to-hex example

If you find that you have numbers for only the first and third places, this means to add a 0 as a placeholder. $040F would be an example.

HEX TO DECIMAL

Conversion in this direction is even easier. Just look up each hex number in the chart, find the corresponding decimal value, place them in a column, and add them up to get a decimal total. Here is the mandatory example (Example 2) you are no doubt expecting:



Example 2. Hex-to-decimal example

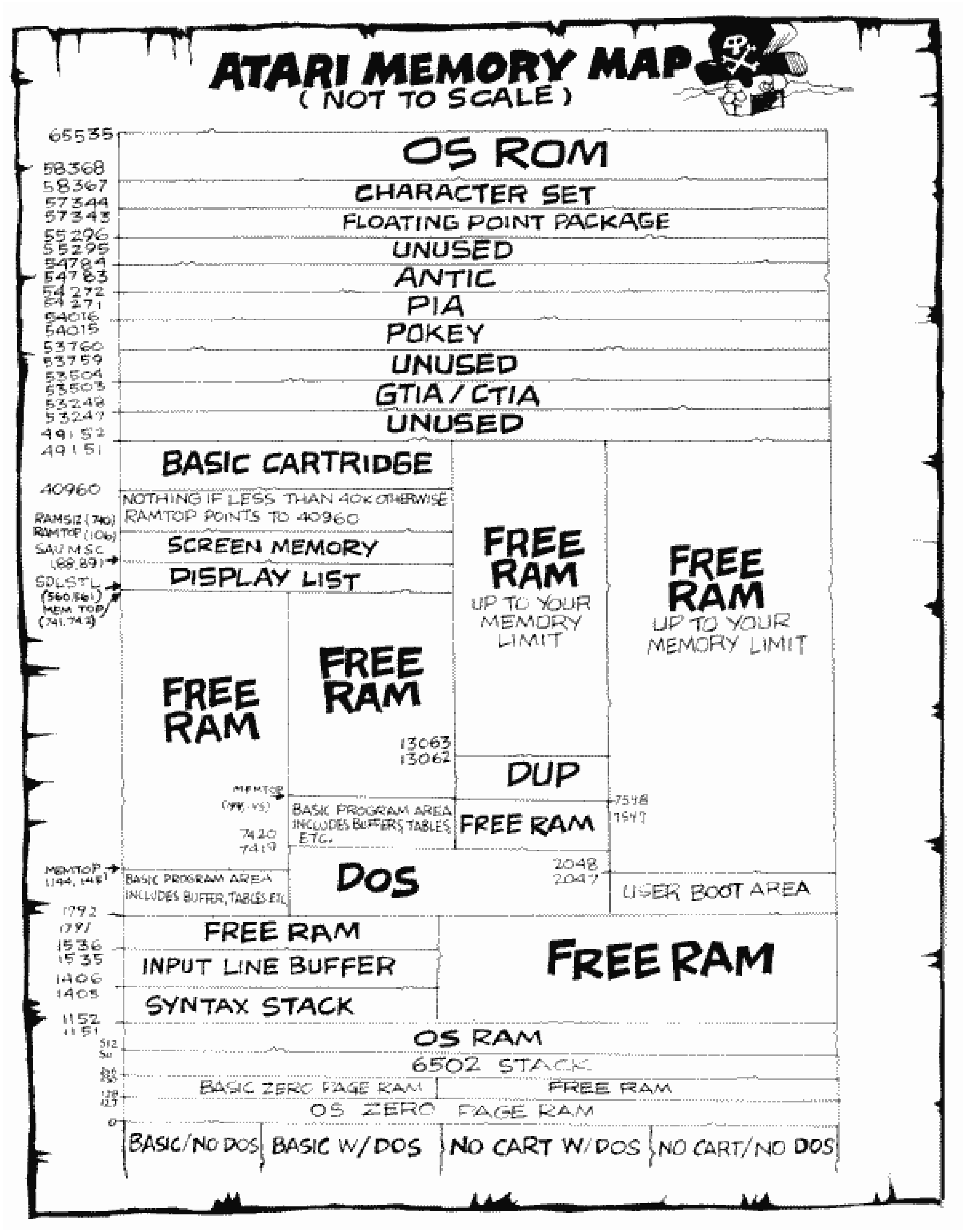


FIGURE 2. Large memory map

HOW TO READ THE MEMORY MAP

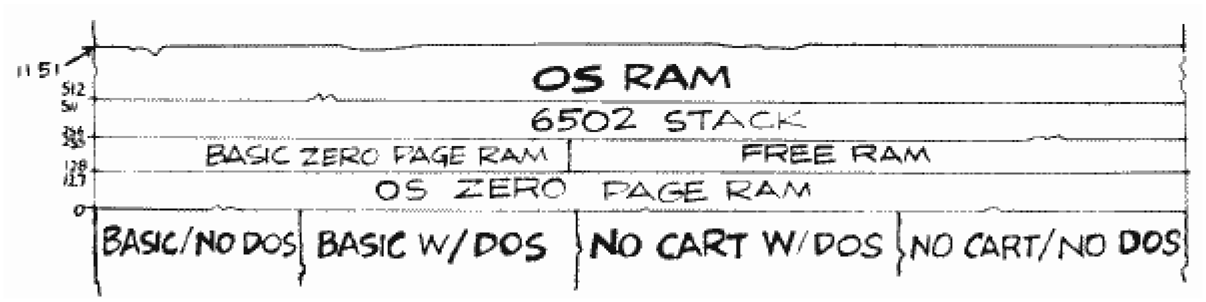
BEGINNING USERS - Read the text that is printed in BOLD TYPE only. What you are reading right now is in bold. These memory locations will be the easiest for you to use and usually don't involve assembly language.

ADVANCED USERS - Read everything! Many areas of memory are not of any practical use, but you can learn a lot about HOW a computer works by reading the boring parts.

The book is formatted like this:

LABEL
DECIMAL # HEXADECIMAL #
DESCRIPTION

HEXadecimal numbers are often preceded by a "$".



PAGE ZERO

Locations 0 to 255 are called "Page Zero" (in the language of computers, a "page" is 256 bytes). Since one byte can hold any number in the range of 0 to 255, the computer only needs one byte to hold the address of a page zero location. This saves time when you have to load or store a value in machine language, so page zero is very important to machine language programs that have to run as quickly as possible. That's why the operating system uses the first 128 bytes. The other 128, locations 128 through 255, can be used by BASIC and you for super fast machine code.

Machine language programmers should note that locations 2 through 7 are not cleared by either a cold start (turning the computer off and then on again) or warm start (pressing SYSTEM RESET) operation.

LINZBS
0,1 0000,0001

The great mystery location! Nobody seems to know exactly what this location does. According to Atari's operating system listing, it is "LINBUG RAM [and] will be replaced by [the] monitor RAM" (your guess is as good as mine), and the only time it uses it is to define it. It does seem to be used to store the VBLANK timer value though, so it's probably not completely useless.

CASINI
2,3 0002,0003

This is used in "cassette initialization." As you probably already know, if you hold down the START button while turning on the computer, the computer will beep. If you then press RETURN, the computer will expect a machine language tape to be in the cassette recorder and will proceed to load it. This process is called "booting" a cassette. The first six bytes stored on the machine language tape contain special information about the tape. The first byte, actually, is ignored. The second tells how many 128 byte "records" are on the tape (when you load in the tape, each beep while loading represents a record). The third and fourth give the starting address memory that the machine code is to be stored at, called the "load" address. The fifth and sixth give the "initialization" address (where to go to get the program set up and ready to run). The initialization address, as you may have guessed, gets stored here at CASINI. Once the whole program has loaded, the computer jumps to the load address plus six (to skip over these special bytes) where the program either tells it to load some more or RTS (ReTurn from Subroutine). When the computer comes across an RTS instruction, it looks in CASINI for the initialization address and JSRs (Jump to SubRoutine) to that address. Finally (and you thought cassette boots were easy), the computer JSRs to the address in DOSVEC (10,11), which gets the program running (DOSVEC should be set up by the program either in the initialization process or as part of a multiple load).

RAMLO
4,5	0004,0005

RAMLO has a bunch of uses, none of which will be useful to you. First, the OS uses it as an index (like the variable in a FOR/NEXT loop) while clearing out memory after you turn on the computer. It also uses it as an index while testing memory to make sure everything is A-OK. Finally, and you'll love this one, it's used to store the "disk boot address," which is usually 1798 in case you care, for the boot continuation routine (which is what happens when you want to load into more than one part of memory). By the way, it's real buddy-buddy with TRAMSZ and TSTDAT (the three work together in the RAM test routine).

TRAMSZ
6	0006

Another location with a whole bunch of uses. As mentioned, TRAMSZ helps out RAMLO in testing the RAM. Its value is then transferred to RAMTOP (location 106). But, before any of that happens, it is used in testing whether or not a left cartridge is plugged in. If there is a left cartridge (also known as cartridge A), then TRAMSZ is set to one. If not, it's set to zero.

TSTDAT
7	0007

This one only has two functions. First, as you already know, it helps out in the RAM test routine (see your OS listing if you're dying to find out what the RAM test routine is). Secondly, like TRAMSIZ, it's used initially in testing whether or not the right, or B, cartridge is present.

Machine language programmers: Locations 8 through 15 are cleared on cold start only.

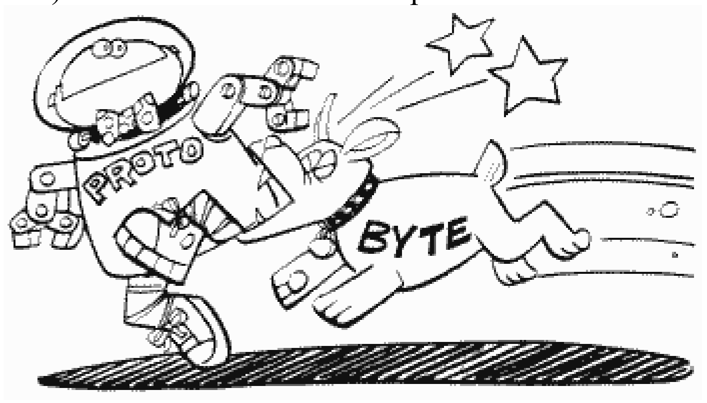
WARMST
8	0008

This is the warm start flag, telling you whether you're in the middle of a warm start or a cold start. If WARMST equals 0, then you're in the middle of a cold start. If it's anything else, then you're in the middle of a warm start (pressing SYSTEM RESET will set WARMST to 255). The main purpose of WARMST is to make sure that if someone presses the SYSTEM RESET button before everything is initialized properly, the computer will know about it and start over instead of messing everything up. Nice stuff to know, but generally useless. But wait, you say, can't I trick the computer into rebooting by changing the value of WARMST to 0, that way preventing people from using SYSTEM RESET to stop my BASIC program so they can LIST it? No. Although you can change the value to 0, as soon as you press SYSTEM RESET it will change back to 255. See location COLDST (580) for a way that you can trick the computer. You might also look at locations POKMSK (16) and STMCUR (138,139) for other ways to protect your BASIC programs from other people's greedy eyes.

Incidentally, warm start normally starts at location 58484.

BOOT?
9 0009

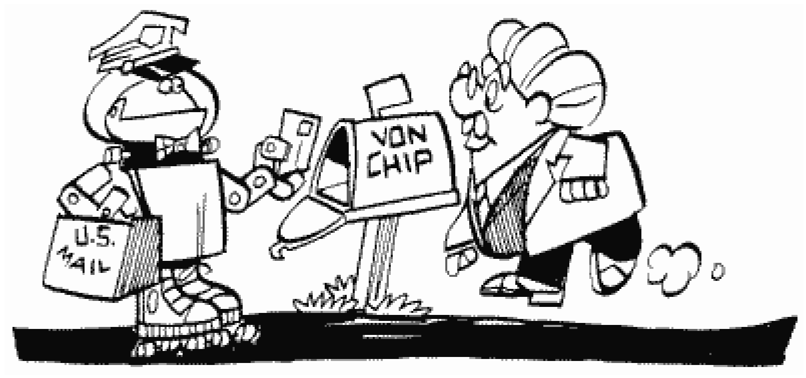
Booting, as you will recall, is the process of loading the program into the computer's memory. In our case the program is loaded from tape or disk. Sometimes a boot is not successful. Maybe you put a rock and roll tape into your Atari recorder by mistake, or you forget to close the disk drive door. In any case, BOOT? is used to tell the operating system whether or not the boot attempt was successful. If BOOT? is equal to one, then there was a successful disk boot. A two indicates a disk boot, and a three (a one plus the two) means both the disk and tape booted. A zero means that everything bit the big one.



If a cassette boot attempt doesn't work, then the OS goes on as though there were no attempt. If the disk boot attempt fails, and this has happened to most of us, then a lovely "BOOT ERROR" message appears on the screen and the OS gives it another try.

OK, now for some miscellaneous stuff. A cassette boot always comes before a disk one. If there is a successful cassette boot, then every time SYSTEM RESET is pressed the computer will go to the address stored in CASINI.

What is an address stored in?



The address is a location where a routine you want to use is located in memory. This address is usually called a VECTOR, because it points to something. You can JSR in machine language or USR in BASIC to get to the routine.

Back to CASINI. If the disk boots successfully, then the computer will go to the address stored in DOSVEC (10, 11). If BOOT? is set to 255 by you, then the computer will "lock up" if SYSTEM RESET is pressed. This is a great way to keep people from looking at your programs. Incidentally, "lock up" means that the computer will not do anything until you turn it off.

DOSVEC
10,11   000A, 000B

This is another vector, used to tell the OS what to do when SYSTEM RESET is pressed. It holds the cassette boot starting address, the disk boot starting address, or the address of the "black-board mode" routine (type 'BYE' from BASIC and press RETURN; that's the blackboard mode and the routine for it starts at location 58481). It's called DOSVEC, because if you're using DOS from BASIC, DOSVEC holds the address that BASIC jumps to when you call DOS (DOS VECtor - get it?). If you want to use this location from BASIC to point to your own routine, then you'll have to make a small change to DOS, since in this case SYSTEM RESET restores DOSVEC to its original value. The change is easy to make, though. All you have to do is POKE 5446 with the Least Significant Byte (LSB) of the address of your routine, and 5447 with the Most Significant Byte (MSB) of the address. The MSB is the first two digits of the hex address, the LSB is the last two. You can compute MSB and LSB from a decimal address with the following formulas: MSB=INT(address/256), LSB=address-(256*MSB). Then call DOS and resave it using the WRITE DOS FILES option. This will give you a custom version of DOS that will allow your routine to run every time SYSTEM RESET is pressed or DOS is called.

Miscellaneous stuff again, DOSVEC is set to 6047, the address of a routine to load in the DUP.SYS file, if DOS is used and it is not told otherwise (i.e., no user boot programs). And, for you machine language dabblers, if you create an AUTORUN.SYS file that doesn't end with an RTS, make sure you set BOOT? to 1 and COLDST (580) to 0 (so as not to confuse the computer).

DOSINI
12,13   000C,000D

This one's easy. Essentially, it is the disk equivalent of CASINI. As a matter of fact, the cassette initialization address is stored here before the OS realizes it's doing a cassette boot and moves it to CASINI. If there is no cassette or disk boot, DOSINI will read 0, 0.

DOSINI can be very useful because it holds the address that the OS jumps to when SYSTEM RESET is pressed. If you have a machine language routine that you want to go to whenever SYSTEM RESET, is pressed, store its address here.

APPMHI
14,15   000E,000F

This location helps prevent your programs from accidentally being written over by the OS. If you're using BASIC, it points to the end of your BASIC program. The OS uses it to determine whether or not there's room for the graphics mode you want to use. As you probably know, the graphics mode stuff (screen memory and display list) is stuck way up at the top of memory. When you tell the OS to set up a graphics mode (with either a GRAPHICS or OPEN "S:" command), it tries to put the display list and screen memory right below the top of memory. Unfortunately, sometimes there isn't enough room, and they would extend down into your program, which you obviously don't want to happen (unless it's a horrible program). APPMHI to the rescue! Before it sets up the requested graphics mode, the OS checks APPMHI to see if there's enough room. If there isn't, it tells you so and sets up a GRAPHICS 0 screen instead, updating MEMTOP (741,742) in the process. MEMTOP, in case you didn't guess, holds the address of the last possible memory location you can use for your program, i.e., the memory location right before the display list. On the other hand, if there is enough room, the desired mode will be set up and MEMTOP updated accordingly.

Sometimes you may want to use the memory between the end of your program and MEMTOP to store character sets, or player/missile information. That's fine, but make sure you change APPMHI so that the OS knows that you're using that memory (in other words, set APPMHI to point to the memory address after the last one you use). Other locations that might be of interest here are CHBASE (54281), PMBASE (54279), and RAMTOP (106).

Machine language programmers: Locations 16 through 127 are cleared on either cold start or warm start.

POKMSK
16 0010

POKMSK is used to turn various types of "interrupts" on or off. An interrupt is exactly what it sounds like; the computer gets interrupted from whatever it's doing and told to do something else (it then usually returns to what it was doing before it was so rudely interrupted).

For machine language programmers, POKMSK deals with POKEY interrupts and is used and altered by the IRQ service. It's also a shadow register for IRQEN (53774).

The following chart (Figure 3) shows exactly what part of POKMSK deals with which interrupts. Change a specific bit to a one to turn on that interrupt, zero to turn it off.

Before we decide whether or not any of this is useful, a few notes for the diehards. The default value for POKMSK is 192; BREAK key and "other key" interrupts enabled.

BIT NO.	DECIMAL VALUE	TYPE OF INTERRUPT
6	64	"Other key"
5	32	Serial input data ready
4	16	Serial output data required
3	8	Serial out transmission finished
2	4	POKEY timer four ('B' and later OS ROMs only)
1	2	POKEY timer two
0	1	POKEY timer one

FIGURE 3. POKMSK chart

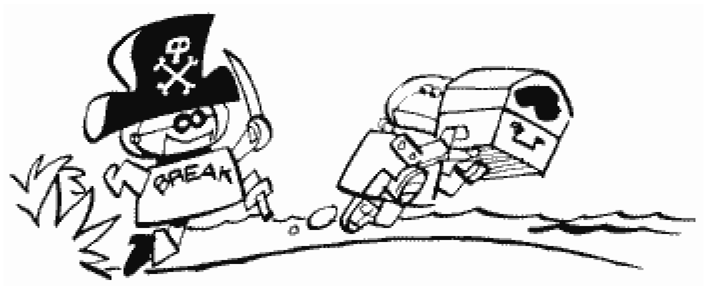
When you enable a timer interrupt, the associated AUDF register will be used as a timer and will generate an interrupt request (IRQ) when it has counted down to zero. See VTIMR1/2/4 (528 to 533) and the POKEY chip (53760 to 54015) for more details.

For you beginners, as well as the pros, there is a handy-dandy use for POKMSK. If you haven't guessed already it allows you to disable the BREAK key so that nobody can BREAK into your program and steal your code. All you have to do is turn bit

seven off. How do you do that? Try the following subroutine:

```
1000 BK = PEEK(16):IF BK > 128 THEN POKE 16,BK -128:POKE 53774,BK -128
1010 RETURN
```

Notice that we also change location 53774. As mentioned before, POKMSK is a shadow register for 53774, and therefore both must be changed. We also check first to make sure that bit seven is on. We do this because, unfortunately, this routine has to be called more than once. You see, the BREAK key is re-enabled by the first PRINT statement that prints to the screen, by an OPEN "S:" or OPEN "E:" statement, by the first PRINT statement after such an OPEN, by the first PRINT statement after a GRAPHICS command, or by a SYSTEM RESET. Phew! To make sure you keep the BREAK key disabled, you'll want to GOSUB to the preceding routine after each such command.



More for the machine language programmer. If you have the newer OS 'B' ROM, there is a vector for the BREAK key interrupt that allows you to write your own routine for the BREAK key. It is called BRKKEY, and can be found at locations 566 and 567.

BRKKEY
17 0011

OK, you've used POKMSK to zonk out the BREAK key. What happens if for some reason you need to know if somebody's pressing it? BRKKEY tells you just that. If it's equal to zero then the BREAK key is pressed (if it's not then it isn't!). If you're looking at BRKKEY from BASIC, remember that you'll have to keep checking it over and over again; BRKKEY tells if BREAK is pressed, not if it were.

Machine language programmers, this location along with POKMSK lets you write your own BREAK key routines if you don't have the 'B' ROM, or if you want to make sure your software will work on the old ROMs. If you do have the 'B' ROM (location 58383 will equal zero if you do), you can use the vector mentioned under POKMSK.

A few boring bits of information. If the BREAK key is pressed during an input/output (I/O) operation, BRKKEY will read 128, not 0. The keyboard, display, screen, and cassette handlers all check BRKKEY to see if they should BREAK (why else?), as do I/O routines and scroll and draw routines. Also look at locations STATUS (48) and DSTAT

(76) for related stuff.

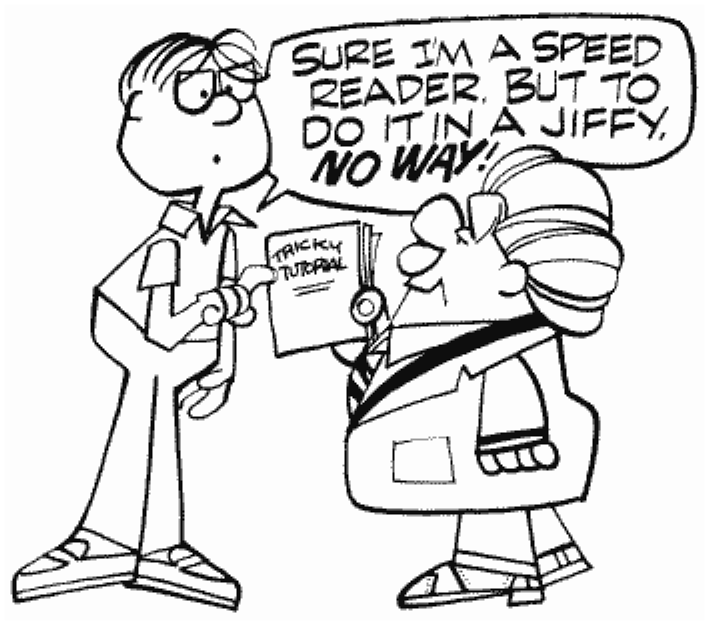
RTCLOK
18-20   0012-0014

This one's actually fun and interesting, and you may even have used it already. It's a clock-the "internal real-time clock" (which just means that it's inside the machine and actually keeps good time). It doesn't count in seconds though, but rather "jiffies." A jiffy is a sixtieth of a second, which happens, not by coincidence, to be the time that it takes the television set to fill the screen. After the screen is filled, a special interrupt occurs, called the Vertical BLANK (VBLANK) interrupt. The OS gets a lot of things done during VBLANK, one of which is updating RTCLOK. Every jiffy (during VBLANK), location 20 gets increased by 1 until it equals 255. At that point, since 255 is the largest number a memory location can hold, it gets reset to 0 during the next VBLANK, and location 19 gets increased by 1. You can probably guess what happens next. When location 19 reaches 255, it gets set to 0 during the next VBLANK and location 18 gets increased by 1. Finally, when location 18 reaches 255, everything gets reset to 0 and the whole thing starts all over again. So, to put things in a more understandable perspective, location 20 increases by 1 every sixtieth of a second, location 19 every 4.27 seconds (256/60), and location 18 every 18.2 minutes (4.27 seconds*256).

The following routine will tell you the number of jiffies, seconds, and minutes that the clock has been running, i.e., since you turned on the computer or last POKEd 18 to 20 with zeros.

```
10 J = PEEK(20) + PEEK(19)*256 + PEEK(20)*256*256
20 S = J/60
30 M = S/60
40 PRINT "RTCLOK reads ";J;" jiffies, or ";S;" seconds, or ";M;" minutes."
```

All three locations are set to zero when you turn on the computer or press SYSTEM RESET. You can set them to whatever values you want just by POKEing them. Possible uses for RTCLOK include timing things that need precise timing. You can even use it to keep track of the time (what an absurd use for a clock).



BUFADR
21,22   0015,0016

This is a temporary register used to store the disk buffer address. It exists so that the OS can use indirect addressing to access the disk buffer. If this doesn't make sense, then BUFADR is not the place for you.

ICCOMT
23   0017

Another hardcore location. ICCOMT holds the CIO (Central Input Output) command and is used as an index into the command table to find the offset for the correct vector to the desired handler routine. Like I said, for hard cores only.

DSKFMS
24,25   0018,0019

This is used as a vector to the FMS (File Management System). It is called JMPTBL by DOS (which doesn't know any better).

DSKUTL
26,27   001A,001B

Another location used by DOS. DOS calls it BUFADR, but we'll continue to call it DSKUTL so as not to get confused with the OS BUFADR (21,22). DSKUTL points to a buffer that the disk utilities package (DUP) uses when copying or duplicating a file. If the user says it's OK to use the program area while copying or duplicating, then DSKUTL gets the value in MEMLO (743,744). If the user says no way to the program area, then

DSKUTL gets the address of DBUF, a special 250-byte buffer at location 7668.

PTIMOT
28 001C

If you're not a big fan of machine language I/O, then skip this one. PTIMOT is the printer timeout value. It's set by your printer handler software, and initialized by the OS to 30, which represents 32 seconds. If you're good at math you'll realize that 60 would represent 64 seconds. It's updated after each printer status request, getting the specific timeout status from DVSTAT+2 (748).

A timeout is essentially what it sounds like. The printer (it could also be a disk drive or similar device) says, "Hey, timeout," and takes five. This has the noticeable effect of the printer just sitting there for a brief period of time doing nothing. Then it decides to come back and get to work again. What are you going to do, fire it? Anyway, those of you with the original OS may be very familiar with this situation, since that version of the OS contained a bug causing unnecessary timeouts. You would be doing something like printing when all of a sudden the computer would stop everything for up to five minutes. Version B did away with it.

PBPNT
29 001D

PBPNT is an index (pointer) into the print buffer. It tells the OS how full or empty the buffer is, and can therefore have any value from zero up to the size of the print buffer, PBUFSZ (30).

PBUFSZ
30 001E

PBUFSZ is the size of the print buffer, but not necessarily the size of the print line. The normal buffer size is 40 bytes (which is obviously not the normal line size for most printers). It is initialized to zero by the OS (and not set until P: is opened), and set to four in the case of a printer status request.

Characters get stored in the print buffer on their way to the printer. The OS checks PBPNT (29) to see whether it's equal to the buffer size (which would mean that the buffer is full) and, if it is, the buffer gets sent to the printer. If the buffer gets an EOL (End Of Line) character, then the OS fills the rest of the buffer with spaces and sends it to the printer.

PTEMP
31 001F

This is used by the printer handler to temporarily hold the character being sent to the printer while it goes off and does some chores.

PAGE ZERO INPUT/OUTPUT CONTROL BLOCK (ZIOCB)

The 16 locations from 32 to 47 are used by CIO to make I/O as efficient as possible (remember the speed advantage of page zero). They are set up the same way as the regular IOCBs (832 to 959) and essentially act as a mirror for the IOCB that wants to be used. In other words, when a CIO operation gets going, the information in the IOCB that's involved is moved to here, where it is used by the CIO routines. When the CIO is all done, then the updated information is moved back to the IOCB. Remember, as complicated as this sounds, it's only done for the sake of speed.

For more information on ZIOCB, CIO, and the rest of the I/O process, see the appendix on I/O.

ICHIDZ
32 0020

This serves as an index into the handler address table for the file that's currently open on this particular IOCB. If there is no such open file (i.e., the IOCB is free), then ICHIDZ gets set to 255.

ICDNOZ
33 0021

The device or drive number. DOS uses it to tell the maximum number of devices, and therefore calls it MAXDEV (I'll bet you can see a connection there). It gets initialized to one.

ICCOMZ
34 0022

This is the command byte, which is set by the user, in the course of setting up the regular IOCB, to tell CIO what kind of operation is to be performed (GET, PUT, FORMAT, etc.). It also determines the format of the rest of the IOCB (which will be different for different commands).

ICSTAZ
35 0023

ICSTAZ is the status of the last IOCB action taken. The device in question tells CIO what happened, CIO tells the OS, and the OS sets ICSTAZ (a little chain of command here). Hopefully everything went OK, but if it didn't, ICSTAZ is the guy who'll know.

ICBALZ,ICBAHZ
36,37   0024,0025

Another buffer address, this one for data transfer. The OS also uses the ICBAZ twins to get the device name from the user (in this case ICBALZ / HZ holds the address of the location where the device name has been stored).

ICPTLZ,ICPTHZ
38,39   0026,0027

Each device has its own routine to "put" a byte into the device. The OS sets this location to hold the address (minus one) of the routine for the device being used. When the file is CLOSEd (and on power up), it is set to the address of CIO's error routine for an illegal put (because you can't put something into a device unless it's open).

ICBLLZ,ICBLHZ
40,41   0028,0029

More buffer stuff. This time we have a counter that is initially set to the maximum number of bytes to PUT or GET in an I/O operation. It gets decremented every time a byte is put or gotten.

Machine language programmers can set this location to the size of the memory block they want to transfer. By checking after each PUT/ GET to see if it's equal to zero, you'll be able to tell when the transfer is done.

ICAX1Z
42   002A

This is the first byte in the OPEN command after the IOCB number. It tells whether the user wants to READ, WRITE, or both.

ICAX2Z
43   002B
OK, the last location was the first byte after the IOCB number, so guess which one this is? Hey, you're on the ball! ICAX2Z has no specific function, it really depends on the device you're using. CIO pretty much uses it as a working variable, although some serial port functions also use it.

Locations 44 to 47 are also called ICSPRZ or ENTVEC and are spare bytes for local CIO usage.

ICAX3Z,ICAX4Z
44,45   002C,002D

BASIC's NOTE and POINT commands use these locations to transfer disk sector numbers.

ICAX5Z
46   002E

CAX3Z/4Z give the sector, ICAX5Z gives the byte within the sector. It is also used to store the IOCB number times 16 (since each IOCB is 16 bytes long, this gives an index to the beginning of the IOCB). In this case, it is called ICIDNO.

ICAX6Z
47   002F

Sometimes this doesn't do anything. But sometimes (only sometimes) it is called CIOCHR and used to temporarily store the byte that's getting ready to be PUT somewhere (aren't computers wonderful?).

EXAMPLES OF USING IOCBs FROM BASIC

(ICAX1Z and ICAX2Z are referred to as AUX1 and AUX2 respectively.

BASIC Command	**Operating System IOCB Parameters**
OPEN #1,12,0,"E:"	**IOCB = 1** **Command = 3 (OPEN)** **AUXI = 12 (READ and WRITE)** **AUX2 = 0** **Buffer Address = ADR("E:")**
GET #1,X	**IOCB = 1** **Command = 7 (Get character)** **Buffer length = 0** **The gotten character is stored in the accumulator.**
Put #1,X	**IOCB = 1** **Command = 11 (Put character)** **Buffer length = 0** **The character is output through the accumulator.**
INPUT #1, A$	**IOCB = 1** **Command = 5 (Get record)** **Buffer length = Len (A$) - 1 (no more than 255)** **Buffer address = Input line buffer**
PRINT #1,A$	**IOCB = 1** **BASIC uses a special put byte vector in the CB to talk directly to the handler.**
XIO 18,#6,12,0,"S:"	**IOCB = 6** **Command = 18 (“fill”)** **AUX1 = 12** **AUX2 = 0**

STATUS
48 0030

A couple of uses for this guy. First, and probably most important (after all, it got its name for this one), it is used to hold the status of the SIO (Serial Input/Output) routine currently taking place. Figure 4 lists known values:

1	($01)	Operation complete (no problems)
138	($8A)	Device timeout (no response)
139	($8B)	Device NAK (no acknowledgement)
140	($8C)	Serial bus input framing error (your guess)
142	($8E)	Serial bus data frame overrun error (worse and worse)
143	($8F)	Serial bus data frame checksum error
144	($90)	Device done error (it packed up shop)

FIGURE 4. Status chart

STATUS also uses TSTAT (793) as a temporary storage location. The other use, you may recall, is as a storage register during SIO routines for the BREAK abort, timeout, and error values.

CHKSUM
49 0031

SIO's data frame checksum. A (much) simplified explanation of checksums is called for here. A checksum is essentially a sum of values used to check that the values were received correctly. When data gets sent somewhere, the computer adds all the values sent into one byte, and then sends that byte as the checksum value. When data is being received, the values are again added and the result compared to the checksum. If the two aren't equal, that means that at least one of the bytes received was incorrect, and the computer usually responds with an error message. In case you're wondering how you can add a whole bunch of bytes together and store the result in just one byte, you can't. If the check sum exceeds 255, then the carry is just added onto it. For example, in the world of checksums, 254+ 31=2,128 + 128 =1, and so on.

A "checksum sent" flag is located at CHKSNT (59). CHKSUM relies on BUFRFL (56) to tell when the checksum is to be sent or received.

BUFRLO,BUFRHI
50,51	0032,0033

Hey, it’s another data buffer! This one is used to hold the stuff that gets sent out or received during I/O. Actually, BUFRLO/HI is a dynamic pointer into the buffer (which just means that it points to the next byte to be sent/received rather than always pointing to the beginning of the buffer).

SIO and the DCB (Device Control Block) both use this pointer.

BFENLO,BFENHI
52,53	0034,0035

A pointer to the byte right after the end of the data buffer described in the previous location. This helps SIO and the DCB determine when the buffer is full.

CRETRY
54	0036

Sometimes you may get an error message trying to do stuff like reading from or formatting the disk. Before you tell the user to go toss the disk in the trash, however, you’ll probably want to double-check to make sure that there really is something wrong with the disk, and it wasn’t just a temporary boo-boo. CRETRY specifies how many times to try again before giving up. It is initialized to 13.

DRETRY
55	0037

The same basic idea as CRETRY, but where CRETRY double-checks that a specific command doesn’t work, DRETRY double-checks to make sure that the whole device doesn’t work. It is initialized to one.

BUFRFL
56	0038

If BUFRFL equals 255, then the data buffer is full. If it doesn’t, it isn’t.

RECVDN
57	0039

If RECVDN equals 255, then all the data that was supposed to be received has been. If it doesn’t, it hasn’t.

XMTDON
58	003A

If XMTDON equals 255, then all the data that was meant to be sent was. If it doesn't, it wasn't.

CHKSNT
59	003B

If CHKSNT equals 255 (you should know this already), then the checksum was sent.

NOCKSM
60	003C

More checksum stuff. A zero here means that a checksum follows the current transmission. No zero means no checksum.

BPTR
61	003D

By now you should be getting the idea that buffers are pretty popular items around a computer. Here's another buffer to further enforce that idea. This time we have one for cassette data. Like BUFRLO/HI, BPTR is actually a pointer into the buffer (which is located at CASBUF [1021 to 1151]), indicating how full or empty the buffer is. It can be anything from zero to the value in BLIM (650). If it's equal to BLIM, then the buffer is either empty or full (depending on whether it was being read into or written out of, respectively). It is initialized to 128.

FTYPE
62	003E

You load in a program from cassette and while it's loading, the computer goes "beeeep (pause) beeeep (pause) etc.," right? Well, the pause has a name. It's called an "inter-record gap." Can you say "inter-record gap"? Sure, I knew you could. Anyway (so much for the comic relief), FTYPE specifies the kind of gap to put on the tape. It equals 0 for normal gaps (like in a CLOAD tape), 128 for continuous (long) gaps (like in an ENTER "C:" tape).

FTYPE gets its value from ICAX2Z (43), which gets it from DAUX2 (779), which gets it from the user.

FEOF
63	003F

OK, we’re still loading from cassette. How do we know when there’s no more to read? The last record (each beep when loading represents a record) on a cassette file has a command byte of 254 and is called the EOF (End Of File) record. FEOF is set to 255 when the EOF record is reached, and 0 before that.

See CASBUF (1021) for an explanation of the way cassette records are structured.

FREQ
64	0040

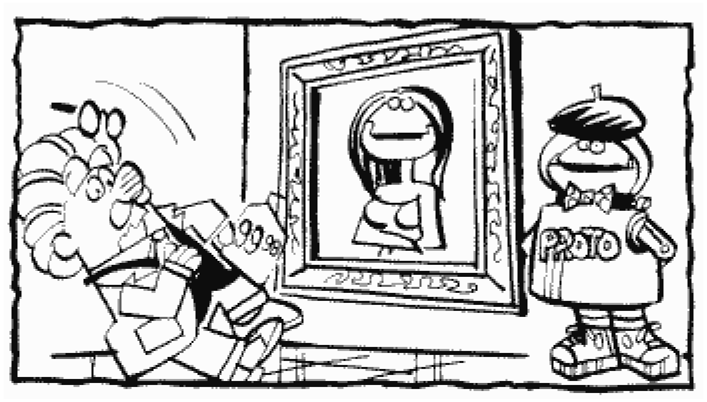
Quite simply, the number of beeps that the Atari makes when you OPEN the cassette handler: one beep for read, two for write (type “CLOAD” and press RETURN for a demonstration).

SOUNDR
**65	0041**

SOUNDR is used to turn the beeping off (or back on) while the cassette or disk program is loading. A zero here will stop the beeping, anything else will get it going again. Also see location PACTL (54018). The beeping is caused by the loading of data from the right channel. Atari added this to the computer so that its educational tapes can talk to you while loading programs. Ah hah! This must mean that the left channel still can be heard even if you change the value in location 65.

CRITIC
66	0042

CRITIC is used to tell the OS that the current I/O operation is time critical (disk or cassette operations, for example). This is important, because in the case of time-critical I/O it is important that the computer spend as little time in vertical blank as possible. When CRITIC is a nonzero value, the OS knows not to execute the second stage of the VBLANK process (CRITIC is checked at the end of stage 1). Since there are some things happening during stage 2 that you may not want to interrupt (check the OS listing if this is really of concern to you), CRITIC should be used only when necessary. To experiment, poke a 2 into 66 and then press any letter. The repeat capability will not work and CONTROL-2 will sound funny. You can’t press any key twice in a row.



The following seven bytes are called FMSZPG and serve as zero page registers for the disk file manager system (FMS).

ZBUFP
67,68   0043,0044

When the FMS does disk I/O, it needs to know the user filename so it can OPEN the file. It expects to find it in a buffer pointed to by ZBUFP.

ZDRVA
69,70   0045,0046

Zero page drive pointer. FMS also uses ZRDVA in its setup, free sector, and get sector routines. I know this sounds somewhat cryptic, but it’s that kind of location.

ZSBA
71,72   0047,0048

A pointer to the sector buffer.

ERRNO
73   0049

If things go wrong during disk I/O, this is where you can find the error number. FMS initializes it to 159.

CKEY
74   004A

If the START button is held down when the Atari is first turned on, CKEY is set to one (zero otherwise). This indicates that a cassette file is to be booted.

CASSBT
75 004B

If a cassette file is booted and the boot is successful, CASSBT gets set to one. Zero means boot no good. Also see BOOT? (9).

DSTAT
76 004C

A location of all trades, DSTAT is used mainly by the display handler to indicate display status and as a keyboard register. It is also used to indicate a cursor out of range error, the BREAK abort status, and too little memory for the desired screen mode.

ATRACT
77 004D

Try leaving the Atari on for about nine minutes without pressing any keys (or save yourself some time by POKEing ATRACT with 128). You've probably run across this effect before; it's called the "attract mode" and, as you can see, causes the colors on the screen to change every four seconds or so, at subdued brightnesses. Why, you may ask? If you leave your computer alone for several hours with a picture on the screen that doesn't change (like when you break for lunch and forget to turn the TV off), it can "burn" the picture tube of your television set and leave a permanent, although faint, image on the screen. You obviously don't want this to happen, so Atari thoughtfully created this solution.

Whenever you press a key, IRQ (Interrupt ReQuest) sets ATRACT to 0. Otherwise, every four seconds VBLANK increments it by 1. When it reaches 127 it gets set to 254 and the Atari enters the attract mode. That's the way it stays until a key is pressed.

The attract mode only changes the four color registers COLPF0 to COLPF3 (53270 to 53273) and the background COLBAK (53274). That means that you'll have to write your own attract routine for DLI induced colors.

If you're using joysticks but not the keyboard, you'll have to set ATRACT to zero every few minutes within your program.

DRKMSK
78 004E

This is one of the two locations used to change the colors in the attract mode (COLRSH is the other). DRKMSK makes sure that the colors aren't too bright. It's normally set to 246 during the attract mode.

For the curious machine language programmers, DRKMSK is ANDed with the original color to mask out part of the brightness nibble. This is done during stage two VBLANK.

COLRSH
79 004F

The other location for changing colors, COLRSH actually does change the colors. It contains the current value of RTCLOK+ 1 (19).

Machine language programmers, COLRSH gets EORed with the color registers (and background) before DRKMSK does its stuff.

Locations 80 to 122 are used by the screen editor and the display handler.

TMPCHR or TEMP
80 0050

Guess what "TEMP" stands for? That's right, this is a TEMPorary (get it?) register used to move data to and from the screen. TEMP gets used by the display handler, which also calls it TMPCHR.

HOLD1
81 0051

Another temporary register for the display handler, this time used to hold the number of entries in the display list.

LMARGN
82   0052

Another tough name to figure out. If you're using graphics mode zero (or have a text window in the mode you're using), LMARGN determines the left margin for text. It's initialized to 2, but you can set it to whatever you want (up to 38). Try POKEing various values into this location.

RMARGN
83   0053

The right margin (I'll bet that somehow you'd figured that out already). It's initialized to 39, and you can also set it to whatever you want (try and set it higher than the left margin though, and less than 40, OK?).

A few words about margins. SYSTEM RESET will restore them to their initial values. Text that is already on the screen will not be affected when you change the margins. Finally, logical lines (the longest a BASIC line can be) couldn't care less where you put the margins. Three lines on the screen and that's it for your logical line, baby, whether that means 120 characters or 3.

ROWCRS
84   0054

This tells you the row on the screen that the cursor is currently on. It works in all the GRAPHICS modes and therefore has a range of 0 to 191 depending on the mode being used. Don't forget that a row is a horizontal line, not a vertical one (you'd be surprised at some of the people that forget). Rows are numbered from top to bottom, 0 being the top.

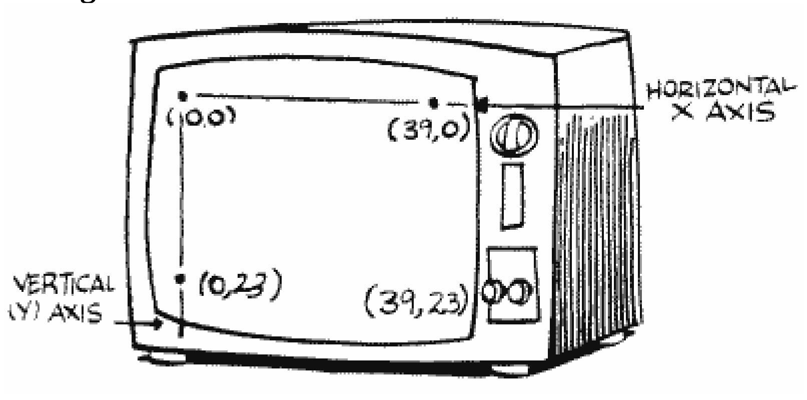
COLCRS
85,86   0055,0056

The column that the cursor is on, ranging from 0 to 319. Location 86 can only get set to 1 in graphics mode 8 (where the column number can exceed 255). Columns are numbered from left to right, 0 being the leftmost column. Incidentally, ROWCRS and COLCRS define the next cursor position to be read or written to, not the last one.

DINDEX
87   0057

This location tells the OS what graphics mode is currently being used (so it knows how to respond to a PLOT or some other screen I/O command). When you OPEN the screen (which the GRAPHICS command takes care of for you), the value of the AUX1 byte is stored in DINDEX. This means that DINDEX can have a meaningful

value of anything from 0 to 11, keeping in mind the GTIA modes are numbered 9 through 11.



Most of the time you'll just leave DINDEX alone, because BASIC takes care of it for you. The times that it does come in handy, however, is when you want to use mixed mode display lists. See the appendix on designing custom display lists for more information on this. It also comes in handy when you want to use the so-called "GRAPHICS 7.5", which gives you twice the resolution of graphics mode 7 with the same number of colors (machine language programmers also know this mode as ANTIC mode "E"). The problem with using this mode is that it is, obviously, halfway between graphics modes 7 and 8. That means that the display list is structured the same as a graphics mode 8 display list, but you have to PLOT to it like it was graphics mode 7. So what, you say? Let's look at an example? The following routine sets up what is called a GRAPHICS 7.5 screen by changing a GRAPHICS 8 display list:

```
100 GRAPHICS 8+16
110 DLIST=PEEK(560)+PEEK(561)*256
120 POKE DLIST+3,78
130 FOR LINE=DLIST+6 TO DLIST+204
140 TYPE=PEEK(LINE)
150 IF TYPE=15 OR TYPE=79 THEN TYPE=TYPE-1
160 POKE LINE,TYPE
170 NEXT LINE
999 GOTO 999
```

A brief explanation of what's going on here. We first set up for a graphics mode 8 screen with no text window. Then we find out where the display list is (see SDLSTL [560,561]) and then change each of the graphics mode 8 commands in it to graphics mode 7.5s. Then, since we have no text window, we must go into a continuous loop or else the screen will switch back to graphics mode 0 (take out line 1000 and see for yourself). RUN the program and you will see the screen go from blue to black as the display list changes. You now have a screen that is 160 dots wide and 192 dots high. Try adding the following lines to the preceding routine:

```
180 COLOR 3
190 PLOT 0,0:DRAWTO 159,191
```

Now RUN the whole thing. Uh-oh! What happened? It's supposed to draw a blue line from the top left corner of the screen all the way down to the bottom right corner. Well, unfortunately the OS still thinks that it's in graphics mode eight, and in graphics mode eight things get plotted differently than we want here. Let's trick the OS into thinking it's in graphics mode seven. That way it'll plot properly (technically speaking, we want two bits to represent a pixel rather than one). Add the following line:

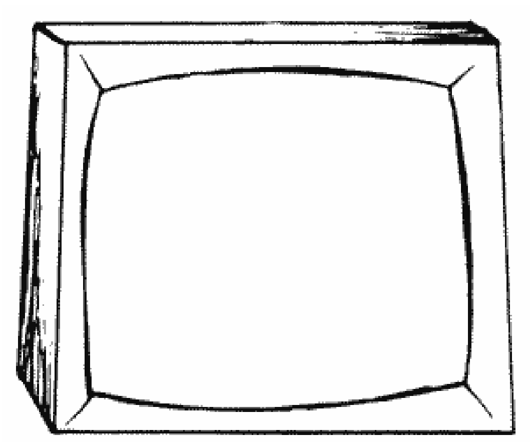
```
175 POKE 87,7
```

RUN it again and whoops! ERROR 141?? That means that the cursor went out of its allowed range. We forgot that graphics mode seven only allows 96 rows. Change line 190 to the following:

```
190 PLOT 0,0:DRAWTO 79,95
```

Now we're OK, but how do we draw in the lower half of the screen? Unfortunately, the tables that tell the OS how many rows and columns each mode has are in ROM, so we can't fool the OS into thinking that there are more rows. The only way around this problem is to treat a GRAPHICS 7.5 screen as being two separate screens, a top and a bottom (machine language programmers can also write their own plot and draw routines). You can use SAVMSC (88,89) to pick the screen you want to use. Try the following program additions and then look at SAVMSC to see what's going on:

```
200 POKE 89,PEEK(89) + 15
210 PLOT 80,0:DRAWTO 159,95
```



(This is a tedious process but it's the price you have to pay if you want the benefits of GRAPHICS 7.5)

SAVMSC
88,89 0058,0059

This is the location of the place in memory where the data is kept that goes onto the screen. Each number in memory represents one character on your TV or several pixels if in a graphics mode. The value at memory location SAVMSC goes on the upper left-hand corner of the screen. The next number in memory goes on the character to the right and so on until the whole row is filled (40 characters in mode 0). The next memory location then goes to the left side, one row down.

When you do I/O to the screen, the OS uses this address to figure out where to PLOT and PRINT. So, for example, the following line will put the letter "A" in the upper left-hand corner of your graphics zero (or one or two) screen.

```
SCRMEM=PEEK(88)+PEEK(89)*256:POKE SCRMEM,33
```

But wait, you say. CHR$(33) doesn't give us an "A"; what's going on here? I'll tell you. The Atari stores the characters in memory in a different order than the ATASCII order (which is what CHR$ uses). See CHRORG (57344) to find out how to convert from one to the other. Anyway, the values in screen memory represent the internal character order rather than the ATASCII one.

If you're not using a text mode, the values you poke to the screen will, obviously, affect the pixels on the screen (the dots on the screen). A pixel is represented by one, two, or four bits. See location DMASK (672) to find out what bits in a byte affect which pixels in each mode (that was easy for me to say). Then try POKEing around. You may want to check CHRORG again; it has an example of using such POKEs to get characters on the screen in graphics mode eight.

OK, so now you know how to change the first character on the screen. What if you want to change the sixth character on the tenth row; how do we know how to find it? Figure 5 shows how many bytes per row are required for each graphics mode.

MODE	0	1	2	3	4	5	6	7	8	9-11
BYTES/ROW	40	20	20	10	10	20	20	40	40	40

FIGURE 5. Number of bytes per row

Now, if you want to change character X in row Y, just multiply Y by the number of bytes per row for the mode you're using and add X (don't forget that the first row and column are numbered zero, not one). Add this value to the address in SAVMSC, and POKE away. For example, let's put the letter "B" in the middle of a graphics zero screen (row 11, column 19):

```
100 GRAPHICS 0
110 SCREEN=PEEK(88)+PEEK(89)*256
120 POS=11*40+19
130 POKE SCREEN+POS,34
```

We want to make sure that we don't try and change a byte that isn't part of the screen, so let's add another line to our chart, this one giving the number of rows in each mode. We'll also multiply the number of rows times and bytes per row to get the total number of bytes taken up by the screen memory (Figure 6).

MODE	0	1	2	3	4	5	6	7	8	9-11
ROWS	24	24	12	24	48	48	96	96	192	192
BYTES	960	480	240	240	480	960	1920	3840	7680	7680

FIGURE 6. Screen memory requirements

Now these values, when added to the address in SAVMSC, will give you the value of the first byte after the end of screen memory. What they don't tell you is how much memory the whole graphics mode takes up. Why not? Because they don't take into account the display list (see SDLSTL [560,561]) and a few bytes that get trapped in the middle of everything. So how do we get this total memory amount? Well, it turns out that RAMTOP (106) points to the top of free memory, which coincides with the first byte after the end of screen memory. MEMTOP (741,742) points to the top of BASIC memory, which coincides with the first byte before the display list. So, if we subtract MEMTOP+1 from RAMTOP*256 (RAMTOP is in terms of pages), we'll get the total memory required. I'll save you the trouble and just give you the values. Our final chart is Figure 7.

MODE	0	1	2	3	4	5	6	7	8	9-11
BYTES/ROW	40	20	20	10	10	20	20	40	40	40
NO. OF ROWS	24	24	12	24	48	48	96	96	192	192
TOTAL SCREEN BYTES	960	480	240	240	480	960	1920	3840	7680	7680
TOTAL MODE BYTES (NORMAL SCREEN)	992	672	420	432	969	1176	2184	4200	8138	8138
(SPLIT SCREEN)	…	674	424	434	694	1174	2174	4190	8112	

FIGURE 7. Screen requirements chart

You may have told yourself by now that you can change the values in SAVMSC and thereby change where the screen is. And if you can change where the screen is, you can keep more than one screen in memory at the same time. Well, you're half right. You definitely can have more than one screen in memory at the same time, but unfortunately SAVMSC only tells the OS where to PRINT and PLOT (and the like) to; it doesn't tell the computer what to display on the television screen. Fortunately, there is another pair of locations that tell what to display, and the word "display" should tip you off to where they are; they're in the display list (this is kind of like adult Sesame Street, isn't it?). Specifically, they're the fifth and sixth bytes in a normal (unaltered by you) display list. Try the following:

```
100 DLIST=PEEK(560)+PEEK(561)*256
110 LOW=PEEK(DLIST+4)
120 LOW=LOW+1
130 IF LOW=256 THEN LOW=0:POKE DLIST+5,PEEK(DLIST+5)+1
140 POKE DLIST+4,LOW
150 FOR DELAY=1 TO 10:NEXT DELAY
160 GOTO 120
```

This will move the starting address of the screen one byte forward at a time, having the effect of swallowing up whatever was on the screen when you ran it. Press SYSTEM RESET to stop it and get everything back to normal.

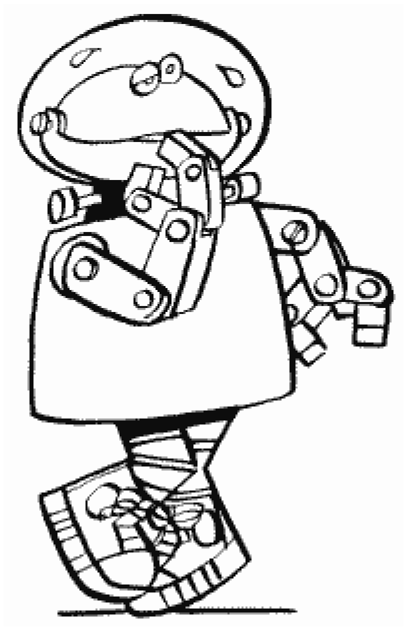
A few things to note here. First, if you let this run for a while (get rid of line 150 to make it happen faster), the screen will suddenly fill up with a whole bunch of garbage. This "garbage" is actually your BASIC cartridge! The starting screen address has been moved so far forward that it has now entered the BASIC zone. You may have astutely noted that the garbage didn't scroll onto the screen smoothly, but rather just sort of suddenly appeared. This is because the screen memory has committed a no-no. It has crossed a 4K boundary. What is a 4K boundary? It's the boundary between one group of 4096 bytes and the next one. How do you tell where one is? Well, first of all the address of a 4K boundary is a multiple of 4096. Better yet, if you're working in hexadecimal, the leftmost digit in the four-digit hex number is the "4K digit" (this is not an official term). When it gets changed, a 4K boundary has been crossed. OK? In any case, the whole purpose of this explanation was simply to tell you that the screen memory is not allowed to cross over a 4K boundary. The GRAPHICS command usually takes care of this for you, but if you're setting up more than one screen, you'll have to be careful.

Going way back to our program example, you should also note that despite what's happening on the TV set, the OS still thinks that the screen is where it was originally, since we haven't changed SAVMSC. If you expect the OS to keep up with you, change SAVMSC as well as the display list.

Finally (and you thought it would never end), before we move onward and upward, a few bits of memory trivia. The address of the text window memory can be found at TXTMSC (660,661). And, in case you thought you weren't going to get a good multiple screen example, you're right. Just kidding.

```
99 REM Get everything set up
100 GRAPHICS 1:PRINT #6;"THIS IS SCREEN ONE"
110 DLIST1L=PEEK(560):DLIST1H=PEEK(561)
120 DLIST1=DLIST1L+DLIST1H*256
130 SCRMEM1L=PEEK(DLIST1+4):SCRMEM1H=PEEK(DLIST1+5)
```

```
140 POKE 106,DLIST1H-4
150 GRAPHICS 2:PRINT #6;"THIS IS SCREEN TWO"
160 DLIST2L=PEEK(560):DLIST2H=PEEK(561)
170 DLIST2=DLIST2L+DLIST2H*256
180 SCRMEM2L=PEEK(DLIST2+4):SCRMEM2H=PEEK(DLIST2+5)
189 REM Do the flipping
190 POKE 560,DLIST1L:POKE 561,DLIST1H
200 POKE 88,SCRMEM1L:POKE 89,SCRMEM1H
210 GOSUB 1000
220 POKE 560,DLIST2L:POKE 561,DLIST2H
230 POKE 88,SCRMEM2L:POKE 89,SCRMEM2H
240 GOSUB 1000
250 GOTO 190
999 REM Pause between screens
1000 FOR PAUSE=1 TO 200:NEXT PAUSE
1010 RETURN
```



Sorry, but no explanation for this one. You should be able to figure it by yourself. I will, however, give you the following lines which you may want to add to make the screen look a little less messy:

```
205 POKE 559,34
235 POKE 559,34
1005 POKE 559,0
```

OLDROW
90 005A

OLDROW is the last row the graphics cursor was on. It gets its value from ROWCRS

(84). DRAWTO and the FILL command (XIO 18) use it to determine their starting row.

OLDCOL
91,92   005B,005C

The last column the graphics cursor was on. Guess where this one gets its value from? You got it, COLCRS (85,86). It gets used the same way as OLDROW does.

OLDCHR
93   005D

When you move the cursor all over the screen, isn't it nice how it doesn't erase characters as it goes over them? Thank this guy for that; OLDCHR holds the value of the character under the cursor so it can be put back when the cursor moves on.

OLDADR
94,95   005E,005F

OLDCHR is great, but the OS has to know where to put it (no suggestions, thank you). OLDADR is the address in screen memory of the current cursor location and is used to help restore the character under the cursor.

NEWROW
96   0060

Out with the OLD and in with the NEW! This is the row that DRAWTO and FILL will draw to or fill to. It is initialized to the value in ROWCRS (84).

NEWCOL
97,98   0061,0062

Same as the preceding except this is the column to draw or fill to and is initialized to the value in COLCRS (85,86).

LOGCOL
99   0063

More cursor position stuff, this time for the benefit of the display handler. LOGCOL tells the position of the cursor within the current logical line. It is equal to the number of rows the logical line has filled so far times 40, plus the current value of COLCRS. Since a logical line can fill up to three rows, this gives LOGCOL a range of 0 to 119.

See BUFCNT (107) for the character length of the logical line.

ADRESS
100,101 0064,0065

A temporary storage location used by the display handler for so many things that it made my mind spin and I forgot what they were.

MLTTMP
102,103 0066,0067

More temporary storage, with aliases OPNTMP and TOADR.

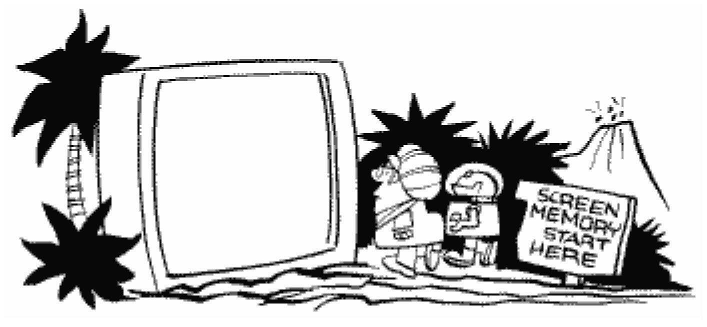
SAVADR
104,105 0068,0069

Also know as FRMADR. Also used for temporary storage. Also not significant enough to explain (look at the OS listing if, for some reason, you really care).

RAMTOP
106 006A

As you probably guessed, this points to the top of RAM. It gets its value from TRAMSZ (6) during the power up operation, as does RAMSIZ (740). Big deal, right? Wrong.

If you're doing custom character sets, player/missile graphics, or anything else where you need a fairly large amount of memory that is safe from BASIC and the OS, RAMTOP is going to save your tush. You see, the OS doesn't care if RAMTOP isn't really the top of memory, so you can change its value and make the OS think that the top of memory is lower than it really is. Then you can go ahead and use the extra locations between RAMTOP and RAMSIZ for whatever you want. It's done something like this:



1. Decide how many pages of memory you want to protect. Remember that a page is 256 bytes (RAMTOP is in terms of pages).

2. POKE RAMTOP with the value in RAMSIZ minus the number of pages.

3. Do a GRAPHICS call. If you don't, your "protected" memory will be in the middle of the screen memory. The GRAPHICS call moves the screen below the new RAMTOP.

4. The locations from RAMTOP (times 256 remember) to RAMSIZ (times 256, minus 1) are now your very own.

Easy, isn't it? Well, not quite. First of all, the first 800 bytes after RAMTOP aren't really safe. The OS scrolls the text window as if it were an entire GRAPHICS zero screen (this saves having to write a special routine for the text window). This means that the OS tries to scroll 20 rows (times 40 bytes per row) after RAMTOP. This is fine normally, because there is no more RAM after RAMTOP that would get messed up. Unfortunately, when you have moved RAMTOP, your RAM is in jeopardy. The first 800 bytes of it, that is. The solution, if you're using a graphics mode with a text window, is just to protect four more pages than you need, and not use the first 1024 (to be safe) after RAMTOP. If you're not using a text window, you still have to protect an extra page, because the first 64 bytes aren't safe for other reasons.

You also have to be careful that the new RAMTOP isn't less than MEMTOP (741,742), since MEMTOP points to the top of your program area.

Confused by "Saving Memory Areas?"

Lastly, because of the 4K boundary problem mentioned under SAVMSC (88,89), you should move RAMTOP by at least 16 pages (16*256=4K) if you're using graphics mode seven or higher.

You can also use MEMLO (743) to protect a different part of memory.

First of all, there is a very simple reason to "protect" an area of memory. If you POKE numbers into memory that currently has nothing in it, then run your program, you may find your data changed when you go to use it. The reason is that BASIC has to move things around as it works. The only way to be sure you have a completely safe area is to move the pointer (106) down so BASIC thinks the top of memory is lower than it really is.

BUFCNT
107 006B

This keeps count of the number of characters currently in the logical line

BUFSTR
108,109 006C,006D

The starting address (in terms of row and column) of the current logical line buffer. It is initialized to the values in ROWCRS and COLCRS when the line is started.

BITMSK
110 006E

The display handler uses BITMSK to mask off bits during the bitmapping process. What? For those of you not into machine language, a dot on the screen (in the graphics modes, not the text modes) is represented in memory by two bits (one bit in graphics mode eight). So, since two bits are only part of a byte, the OS has to have a way of changing bits without changing the other parts of the byte. The process it uses is called "masking," and uses the AND and ORA assembly language commands to clear and set individual bits respectively. See a book on assembly language for a more detailed explanation of these two commands. Bit mapping, by the way, refers to the whole process of manipulating the bits to come up with the desired graphics.

SHFAMT
111 006F

Masking (explained in the previous location description) can be a pain in the byte (sorry). The problem is not in the actual ANDing and ORAing, but rather in getting the bits ready to be masked into the byte, or dealing with them after they've been masked out. Think about it for a second. In graphics mode eight, for example, each bit represents a dot on the screen (called a "pixel"). That means that once you mask out the bit you're interested in, there are eight possible positions it could be in. You obviously don't want to have to write the code to deal with eight different cases. Well, Atari didn't either, so they came up with SHFAMT. SHFAMT is used to shift the bits to the right, one bit at a time, until the bits you are interested in are all the way over to the right (right justified). It's easier to deal with them there. Once you're done having your way with them, SHFAMT helps you get them back to their proper places.

OK, we've got a cute explanation, but what's really going on? SHFAMT initially gets the value in DMASK (672), which is used to mask out the desired pixels. SHFAMT is then shifted to the right (LSR) one bit. If a bit hasn't fallen out of the byte in the process (the carry flag is clear), the masked-out bits are also shifted to the right one bit and the whole thing is repeated. If a bit did fall out, then the masked-out bits are right justified. To get the bits back to their proper position, SHFAMT is restored to the value in DMASK and the same thing happens except this time the masked-out bits are shifted to the left (ASL) one bit at a time (SHFAMT is still shifted to the right though). Then the bits are ready to be masked back into the display byte.

This is a very important and powerful process to understand if you're doing your own bit mapping. Check your OS listing and DMASK for more details.

ROWAC
112,113 0070,0071

ROWAC, along with COLAC (next), are essentially graphic workspaces, used primarily in the "what point do we plot next?" process. ROWAC, of course, is used in the row calculations.

COLAC
114,115 0072,0073

Used in column calculations for point plotting. See ROWAC.

ENDPT
116,117 0074,0075

ENDPT is initialized to either the value in DELTAR (118) or the one in DELTAC (119,120), depending on which is larger (it gets the larger of the two). It is then used to figure out when the final row or column in the line we're drawing has been reached.

DELTAR
118 0076

DELTAR is the absolute value of the difference between OLDROW (90) and NEWROW (96). In other words, it's the number of rows we're going to be drawing across.

DELTAC
119,120 0077,0078

The number of columns we're going to be drawing across. Determined by subtracting OLDCOL (91,92) from NEWCOL (97,98) and taking the absolute value.

ROWINC
121 0079

When the OS computed DELTAR above, it took the absolute value of the result of NEWROW minus OLDROW. The sign of this result, however, is also important to us because it tells the direction we'll be drawing in. ROWINC is one if the sign was negative (we'll be drawing up), and 255, (which also equals minus one in two's complement arithmetic) if it was positive (we'll be drawing down).

COLINC
122 007A

If NEWCOL minus OLDCOL is negative (we'll be drawing left), COLINC is set to one. If it's positive (we'll be drawing right), COLINC is set to 255.

Note that together DELTAR, DELTAC, ROWINC, and COLINC define the slope of the line to be drawn.

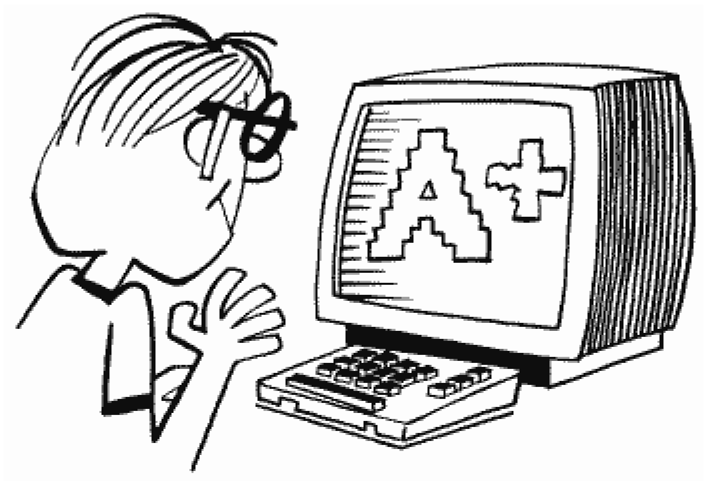
SWPFLG
123 007B

If you're using a split screen mode, it's easier for the OS to print to the text window if it has all the cursor information for it in locations 84 to 95. But it also has to remember the cursor information for the main part of the screen, so what it does is swap the two. SWPFLG equals zero if they haven't been swapped, 255 if they have.

The text window information is kept in locations 656 to 667.

HOLDCH
124 007C

A character that has been typed in from the keyboard goes here so the OS can check out just what kind of character it really is (CTRL, SHIFT, etc.).



INSDAT
125 007D

More display handler temporary storage. I'll even tell you what it's used for; it holds the character under the cursor and is used for end of line detection. Wasn't that exciting?

COUNTR
126,127 007E,007F

Well, here we are back at drawing a line. COUNTR tells how many points have to be plotted before the line is finished. It starts off with the same value as ENDPT (116,117). Then, every time a point is plotted on the line, it gets decremented by one. When it gets all the way down to zero, the line is finished and we can all go home.

The remaining zero page locations (128 to 255) are used by BASIC, with some free for your use. The breakdown looks something like figure 8.

128-145	($0080-$0091)		BASIC program pointers
146-202	($0092-$00CA)		Various BASIC locations
203-209	($00CB-$00D1)		Free for your use
210-211	($00D2-$00D3)		Reserved for use by BASIC
212-255	($00D4-$00FF)		Used for floating-point arithmetic

FIGURE 8. Location 128-255 breakdown

If you're using a language other than BASIC, check its instruction manual to find out which of these locations it uses.

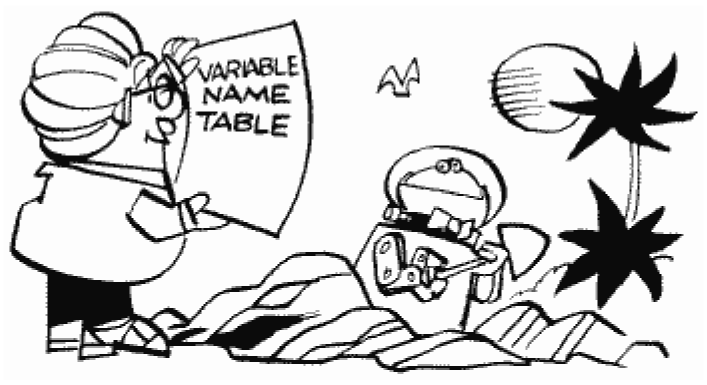
If you are programming in machine language, and not using a cartridge, all 128 bytes here are probably free for your use. Check your assembler's manual.

LOMEM
128,129 0080,0081

LOMEM points to the beginning of the RAM available for BASIC programs (in other words the end of the OS RAM). It gets the same value as MEMLO (743,744) initially, and every time the BASIC "NEW" command is used. Although this implies that its value can differ from that of MEMLO, this doesn't seem to be the case. The only difference between LOMEM and MEMLO appears to be that BASIC uses LOMEM while the OS uses MEMLO.

The first 256 bytes after LOMEM are used as a buffer by BASIC for the tokenization process. Tokenization refers to the process of taking your program and scrunching it up so it takes up as little space as possible. Essentially, each command and variable is replaced with a number (called a "token"). That way, it only takes one byte to store a command, rather than one byte for each letter in the command (this is an extremely simplified description; see De Re Atari for a complete play-by-play). Note that the SAVE command saves the program in tokenized form, while LIST saves it just the way you typed it in. That's why a SAVEd program will be shorter than a LISTed one. Incidentally, if you SAVE a program, the values in locations 128 to 141 are saved along with the program.

BASIC also uses the buffer as a stack to evaluate expressions (8+2 is an expression), in which case it calls it ARGOPS. See RUNSTK (142,143) for a description of stacks.



VNTP
130,131 0082,0083

This points to the table where the variable names are kept. The variable names are stored in the order they were typed (which is not the same as the order the program uses them) in ATASCII. To mark the end of a variable name (so you know when the next one starts), the last character of each variable (a letter or digit for regular variables, a "$" for string variables and a "(" for arrays) is stored in inverse video

(add 128 to the ATASCII value of the character). Enough talk, here’s an example of how to print the variable list:

```
100 VNTP=PEEK(130)+PEEK(131)*256
110 VNTD=PEEK(132)+PEEK(133)*256
120 FOR LP=VNTP TO VNTD-1
130 CH=PEEK(LP)
140 IF CH>127 THEN PRINT CHR$(CH-128);" ";:GOTO 160
150 PRINT CHR$(CH);
160 NEXT LP
```

VNTD (next location), of course, holds the address of the end of the variable name table (plus one).

There are a few other useful things you should know about the variable name table. First of all, if you used any variables while you were writing your program (including those used in the immediate mode) but don’t use them now that the program is done, they’re still in the table taking up space. In order to get rid of it, you must LIST your program to disk or cassette, type NEW, and then ENTER the program back in (this has the effect of typing in the final version of the program from scratch).

Second, you can have up to 128 different variables in your program. When BASIC tokenizes the program (see LOMEM), it replaces each variable name with a number equal to the position of the variable in the name table plus 128 (128 if it’s the first variable in the table, 129 if it’s the second, and so forth). This saves a lot of memory.

Third and last, there’s a neat trick you can use to make your program look like garbage when it’s listed. All you have to do is change all your variable names to a RETURN character. This will protect your programs from being looked at by others. The following routine will do it for you. You can’t get things back to normal, so make sure you have an original version of your program saved before you try this:

```
30000 VNTP=PEEK(130)+PEEK(131)*256
30010 VNTD=PEEK(132)+PEEK(133)*256
30020 FOR LP=VNTP TO VNTD
30030 POKE LP,155
30040 NEXT LP
```

VNTD
132,133 0084,0085

The address of the first byte after the end of the variable name table.

VVTP
134,135 0086,0087

Now we know where the variable names are stored and we're about to find out where the variable values are stored. VVTP, you see, points to the variable value table (we'll call it the VVT).

The Atari has three different kinds of variables. There are the scalars (like X, HI, and FUNSTUFF),the arrays (like JULY (4) and SWEET (16,2)), and the strings (like MONEY$). Each of these has a different representation in the VVT, but they all take up eight bytes per variable. Let's take a look at how those bytes are used:

The first byte tells what kind of variable it is. Scalars get a 0, arrays a 65, and strings a 129. Actually, if you forgot to DIMension the array or string in the program (shame on you), you can knock one off the preceding value given above.

The second byte tells what variable name we're talking about here. It's the position of the variable in the variable name table (0 for the first variable, 1 for the second, and so on up to a maximum of 127).

If we're dealing with a scalar, the remaining six bytes give its value in Binary Coded Decimal (BCD). I suspect a quick explanation is necessary here. BCD, as the name implies, is a way of storing a decimal number in binary. Everything alright so far? Good. Atari doesn't seem to follow the standard 6502 BCD format, so I'll give the Atari breakdown. The first byte is the exponent; 64 means 0, 63 means minus 2 (65 for plus 2, 66 for plus 4) and so forth. Add 128 if the value of the variable is negative. The second byte gives the two decimal digits to the left of the decimal point (in BCD, the upper nibble gives one digit, the lower nibble gives a second). The last four bytes give the eight digits to the right of the decimal point. If this makes no sense to you, look up BCD in any introductory book on machine language programming. It probably still won't make sense.

Back to the VVT. If the variable isn't a scalar (after the preceding description, pray that it isn't), then the third and fourth bytes give an offset into the string/array area (see STARP [140,141]). This offset points to the beginning of the data for that variable (relative to the beginning of the string/array area, of course).

If it's an array we're dealing with, the fifth and sixth bytes give the first dimension and the seventh and eighth give the second. No BCD here, just plain old binary. In case you're wondering what I'm talking about, a dimension is the number(s) plus one you use in BASIC's DIM statement. For example, the first dimension in DIM A(5,7) is six, and the second is eight. The reason that one is added is because A(0,0) is a valid array element and, therefore, the array in our example is actually six elements by eight, not five by seven.

If it's not an array (and it wasn't a scalar), then it must be a string. In that case the

fifth and sixth bytes give its current length and the seventh and eighth its DIMensioned length (up to 32767).

Note that the value of VVTP will change every time a new variable is added.

STMTAB
136,137 0088,0089

The variable names and their values are all set, now where's the program? STMTAB tells you just that. It holds the address of the statement table, which is just a fancy name for your tokenized program (plus the last line you typed in without a line number, called the "immediate mode line").

The statement table contains each of the tokenized lines, one after the other. As I mentioned earlier, you should see De Re Atari for a complete description of the tokenization process (which takes place in a buffer pointed to by LOMEM [128,129]). I will, however, fill you in on a few useful tidbits of information.

The first two bytes of each tokenized line give you the line number (in binary). The immediate mode line has a line number of 32768. The third byte tells you the number of bytes from the beginning of this line to the beginning of the next line. The fourth byte tells you the number of bytes from the start of the line to the start of the next statement (in case you use the ":" to put more than one statement on a line), and that's all you'll get out of me.

Try the following to tell you how many lines you have in your program:

```
30000 STMTAB=PEEK(136)+PEEK(137)*256
30010 LINES=0
30020 LINENO=PEEK(STMTAB)+PEEK(STMTAB+1)*256
30030 IF LINENO=30000 THEN PRINT "Your program has ";
LINES;" lines.":END
30040 LINES=LINES+1
30050 STMTAB=STMTAB+PEEK(STMTAB+2)
30060 GOTO 30020
```

STMCUR
138,139 008A,008B

STMCUR is a pointer into the statement, which BASIC relies on when it needs to refer to particular tokens while processing a line in the statement table. When a program isn't running, and BASIC is just sitting around, it points to the beginning of the immediate mode line.

Try the following to create a program that can't be LOADed, only RUN:

```
32767 POKE PEEK(138)+PEEK(139)*256+2,0:SAVE "D:RUNONLY":NEW
```

You can use any filename, of course (and can substitute “C:” for “D:” if you’re using cassette). Make sure this is the last statement in your program. If you want, you can include the routine for changing variable names (see VNTP [130,131]) right before this line to further protect your program.

To use the routine, GOTO 32767. Then RUN "D:RUNONLY" or RUN "C:RUNONLY" (substitute your filename for RUNONLY).

STARP
140,141 008C,008D

STARP holds the address of the string/array area, which is where all the string characters and array values are stored (see VVTP [134,135] to find out how to determine where each variable is within this area). It also points to the end of your BASIC program, which should hint to you that its value will change as your program changes.

Array values are stored in six-byte BCD form (see VVTP [134,135]), while strings use one byte per character. If you DIMension an array such as A(x), where x is the number of elements in the array, then it will take up x*6 bytes in the string/array area, regardless of how many of the elements you use. The same goes for strings. If you DIMension ANS$(y), then y bytes will always be reserved for it in the string*array area, even if you never use it. For this reason, you should be careful when DIMensioning variables and should also make sure that all unused variables are removed from the final version of your program (see VNTP [130,131]).

A few bits of miscellanea. The beginning address in the string/array area of the data for a string is the same as the address you get with the ADR function. More importantly, there is a way you can save a lot of memory using STARP. Here’s the scoop.

A lot of times our programs have strings or arrays in them that always get initialized to the same lengths and values. It may be a string that holds a redefined character set or a machine language routine, for example. Anyway, somewhere in your program you have an initialization routine and the data for the string or array, right? Well, you just found out that the data is also stored in the string/array area. That means that it's in memory twice (the other time is in the tokenized program listing). That's very bad, and I'm going to show you how to do something about it.

As mentioned, STARP also points to the end of your BASIC program. What happens if we change STARP so that it points to the end of the data for the strings/arrays in question? BASIC will think it's part of the program, which means we can SAVE that part of the string/array area with the program! And that means no more need for initialization, so we can get rid of the initialization part of the program. Here's an example of how to do it:

```
99 GOTO 200:REM You should GOTO 100 the first time through
100 DIM TEST$ (32)
110 TEST$="We'll save this with the program"
120 STARP=PEEK(140)+PEEK(141)*256
130 NWSTARP=STARP+32
140 HIGH=INT(NWSTARP/256):LOW=NWSTARP-256*HIGH
150 POKE 140,LOW:POKE 141,HIGH
160 SAVE "D:TEST.BAS"
170 STOP
200 STARP=PEEK(140)+PEEK(141)*256
210 NWSTARP=STARP-32
220 HIGH=INT(NWSTARP/256):LOW=NWSTARP-256*HIGH
230 POKE 140,LOW:POKE 141,HIGH
240 POKE 142,LOW:POKE 143,HIGH
```

```
250 POKE 144,LOW:POKE 145,HIGH
260 DIM TEST$(32)
270 TEST$(32,32)="m"
280 PRINT TEST$
290 STOP
```

You're probably wondering how to use this monstrosity, so I'll be a nice guy and tell you. There are two basic parts to it. The first, lines 100 to 170, initialize the string, move STARP to the end of the string, and save everything to disk (you can use C: as well). You could alternately get rid of these lines right before you save the program, because they won't be necessary any more. The second part, from line 200 on, restores STARP and a few other locations that were affected, redimensions the string, and sets the last character so that BASIC knows how long the string is. Now we can print TEST$ and verify that it was indeed saved with the program!

Ok, now how do you adapt this to your own program? First of all, make sure the strings/arrays you want to save are the first variables you use in the program (use VNTP to get rid of unused variables). DIMension and initialize them (you can use a GOSUB to the initialization; it doesn't have to be at the beginning of the program). Now figure out how much memory they take up: one byte for each character, six for each array element. Add this to the current value in STARP and store the new value back in STARP. STOP the program. Get rid of the part of the program that did all of the preceding stuff (including the part for initializing). Add lines 200 to 270 at the beginning of the program (making the appropriate changes in lines 210, 260, and 270) and then save it to disk. That's it.

One last tidbit. There is a quick, easy, little known way of filling a string variable with the same character. It works because of the way BASIC is written. Try this:

```
100 DIM FILL$(800)
110 FILL$(1)="F"
120 FILL$(800)="F"
130 FILL$(2)=FILL$
140 PRINT FILL$
```

RUNSTK
142,143 008E,008F

This one is a pointer to the runtime stack. What is a "runtime stack"? Let's start off with a quick explanation of a stack.

Every seen a stack of trays in a cafeteria? Customers take trays off the top, cafeteria people put trays on the top, if you're not lucky there'll be a mad rush of people and by the time you get to the stack there are none left and the cafeteria people are nowhere to be seen. Well, a computer stack is the same thing, except it uses memory locations instead of trays and there are no cafeteria people. A special memory location is used to point to the current top of the stack.

Now you know what a stack is, so let's talk about the runtime stack. Runtime just means that it's used while the program is running. When you use a GOSUB or a FOR/NEXT loop, BASIC has to be able to remember certain things, so it puts them on the stack until it needs to refresh its memory. Now you need to know what exactly gets put on the stack.

For each GOSUB encountered, four bytes are put on the stack (they are taken off when BASIC RETURNS from the subroutine). The first byte is a zero and tells BASIC that this is a GOSUB. The second and third give the line number that the GOSUB was on, and the last one is an offset into the line so that BASIC knows where to continue from after the RETURN.

FOR/NEXT loops are a little more complicated; they require 16 bytes to be put on the stack. The first 6 bytes give the number (in BCD) that the counter in the loop can go up TO. The second 6 give the STEP value (also in BCD). The thirteenth byte gives the variable number plus 128 of the counter variable. The next two give the line number that the FOR statement was on, and the last one gives the offset within that line of the FOR. These values remain on the stack until the FOR/NEXT loop is complete.

There is one exception to the preceding two paragraphs. A BASIC POP statement will take the top entry off the stack, be it a GOSUB or a FOR/NEXT. You should make sure you POP the stack if you have to leave a FOR/NEXT loop before it's finished or a GOSUB before the RETURN.

Don't forget that the stack is constantly changing, so its size will vary.

Lastly, since the beginning of the runtime stack is also the end of the string/array area, BASIC also calls it ENDSTAR. OK?

MEMTOP
144,145 0090,0091

Two uses for this one. First, relevant to the last location, MEMTOP is also called TOPSTK and points to the end of the runtime stack. Since the runtime stack is the last section of memory used by your BASIC program, MEMTOP is a pointer to the end of your BASIC program (which makes sense, right?). The memory locations from the address in MEMTOP plus one, all the way up to the display list (see SDLSTL [560,561]), are free for your use (but don't forget that the value in MEMTOP will change during program execution, since the runtime stack will be growing and shrinking).

For those of you who are still alert, don't confuse this MEMTOP with the MEMTOP at 741 and 742. This is the BASIC MEMTOP; the other is the OS MEMTOP.

The BASIC cartridge uses locations 146 to 202 for various uses, not all of which are

worthwhile reporting on. With the following exceptions, of course.

FORLN
160,161   00A0,00A1

FORLN holds the line number of the current FOR statement encountered. For example,

```
100 FOR X=1 TO 25
110 NEXT X
120 PRINT PEEK(160)+PEEK(161)*256
```

LSTPNT
173,174   00AD,00AE

List pointer. Contains the location of the line being LISTed. When you just type LIST, you find 32767 here.

DATLN
182   00B6

Points to the number of the item within the DATA statement. This means we are currently reading the first number, the second, etc. Try this program:

```
10 FOR I=1 TO 8
20 READ A
30 ? PEEK(182)
40 NEXT I
50 DATA 1,2,3,4,5,6,7,8
```

DATALN
183,184   00B7,00B8

DATALN holds the line number of the DATA statement that was last READ. For example,

```
100 READ A
110 PRINT PEEK(183)+PEEK(184)*256
1000 DATA 10
```

You can use DATALN in an error-trapping routine to find out where a READ error occurred.

STOPLN
186,187   00BA,00BB

STOPLN holds the line number that the program was on when the program

STOPped, the BREAK key was pressed, or an error was TRAPped. It is also useful in error trapping routines. Now for our example:

```
100 TRAP 30000
110 NEXT Y
30000 PRINT PEEK(186)+PEEK(187)*256
```

ERRSAV
195 00C3

This location holds the number of the error that was TRAPped or caused the program to stop. Try this:

```
10 TRAP 100
20 REM THE REST OF YOUR PROGRAM
.
.
.
100 ? "ERROR #"; PEEK(195):LIST(PEEK(186)+256*PEEK(187))
```

PTABW
201 00C9

When you print a whole bunch of items, and separate them by commas in the PRINT statement (like PRINT A,B,C$), they get printed on the screen with a bunch of spaces in between them, right? Well, PTABW tells how many spaces to separate them by. In technical terms, that means it tells how many spaces there are between each tab stop on the screen (see TABMAP [675 to 689] if you want to set tabs for the TAB key). It can be set to any value from 3 to 255 but is initialized to 10. Let's look at an example:

```
100 PRINT 1,2,3
110 POKE 201,5
120 PRINT 1,2,3
```

SYSTEM RESET doesn't restore PTABW to its original value, GRAPHICS doesn't, nothing does. This is a very durable location.

POKEing a zero here will cause the Atari to lock up shop when it encounters a comma in a PRINT statement.

BININT
202 00CA

If you put anything other than a zero here, then going into the immediate mode (i.e. SYSTEM RESET, BREAK, or the program ending) causes the program currently in memory to erase itself-yet another fun way to prevent people from looking at your program (I personally like this one; it's devious).

```
100 POKE 202,1
110 PRINT "Now try LISTing this program"
```

Noname
203-209 00CB-00D1

These locations are free, free, free for your use if you're programming in BASIC. If you're using a different language, check the accompanying documentation to find out which page zero locations it leaves free.

Noname
210,211 00D2,00D3

These two locations are reserved for BASIC, which means they have no specific use but you should stay away from them.

THE FLOATING POINT PACKAGE

The remaining page zero locations from 212 to 255 are used by the OS's floating point package, a whole bunch of subroutines that BASIC uses when doing math and that kind of stuff. The routines themselves are stored in the OS ROM, so if you don't use them at all in your program, these locations will be free. Don't count on it though, even if you think you're not using the routines. They can sneak up on you when you least expect it.

Floating point math uses six-byte BCD, which was explained briefly under location VVTP (134,135). See the section in De Re Atari on the floating point package for more information.

Unfortunately, the listing for the floating point package is mighty hard to come by, so some of these locations are going to have real short explanations. My apologies to you, and my thanks to the OS Manual and Mapping the Atari for the information I couldn't find anywhere else.

FR0
212-217 00D4-00D9

Floating point register zero. A floating point register is just a place used to hold floating point numbers while operations are performed on them (it may also hold a partial result of an operation). They are all, including FR0 of course, six bytes long since they must hold a six byte BCD representation of the number.

FR0 is also used by the USR command. Remember that USR has the format X=USR(address [,argument][…]) where X can be any variable and the arguments are optional. If you want your machine language routine to give a value to X, you should store that value in the first two bytes of FR0 (212,213 - low byte and high byte respectively) before your RTS statement. BASIC will automatically convert these bytes into a floating point number and store it in X (or whatever variable you used for the call). If you're not using BASIC, you can use FR0 yourself to convert binary values to floating point and vice versa. Put the binary number in locations $D4 and $D5 and then JSR $D9AA to convert to floating point (the result will be store in FR0). To convert back, JSR $D9D2. Note that you can't use these routines from BASIC since BASIC is constantly using FR0 and will mess up your values.

FRE
218-223 00DA-00DF

This isn't very well documented, but it appears to be an extra floating point register.

FR1
224-229 00E0-00E5

Floating point register one. FR1 has the same format as FR0 and is often used in conjunction with it, especially for two-number arithmetic.

FR2
230-235 00E6-00EB

Floating point register two.

FRX
236 00EC

A single-byte register used for single-byte calculations.

EEXP
237 00ED

The value of the exponent (E). I suspect this is the exponent of the floating point number currently being processed, but this is only a suspicion.

NSIGN
238 00EE

The sign of the floating point number (same suspicion as above).

ESIGN
239 00EF

The sign of the exponent in EEXP (237).

FCHRFLG
240 00F0

The first character flag. Your guess is as good as mine.

DIGRT
241 00F1

The number of digits to the right of the decimal point (zero to eight).

CIX
242 00F2

An offset into the text buffer pointed to by INBUFF.

INBUFF
243,244 00F3,00F4

Finally something that can be understood! There are times when BASIC has to convert an ATASCII representation of a number to the corresponding floating point value (like when you type in X =1000). INBUFF points to a buffer used to hold the ATASCII representation. The result gets stored in FR0.

See LBUFF (1408 to 1535) for the buffer itself.

ZTEMP1
245,246 00F5,00F6

A temporary register.

ZTEMP4
247,248 00F7,00F8

Another temporary register.

ZTEMP3
249,250 00F9,00FA

Still another temporary register (will it never end?).

RADFLG
251	00FB

RADFLG determines whether the trigonometric functions (SIN, COS, etc.) are performed in radians or degrees. If it’s zero, then radians are used. If it’s six, then degrees are in fashion. SYSTEM RESET and NEW both restore RADFLG to radians (zero).

BASIC also calls this location DEGFLG.

FLPTR
252,253	00FC,00FD

FLPTR holds the address of the floating point number that the package is now operating on. FLPTR and FPTR2 (to follow) point to the addresses where the results of the current operation will be stored. The documentation is sketchy though, so I’m just making an educated guess.

FPTR2
254,255	00FE,00FF

FPTR2 holds the address of the second floating point number that the package is operating on.

PAGE ONE

Locations 256 to 511 are called page one and have a very important use. They make up the stack for the OS, BASIC, and DOS (see RUNSTK at locations 142 and 143 for an explanation of what a stack is). On power up (and on SYSTEM RESET), the stack pointer is set to 511. Each time a machine language JSR or PHA (PusH Accumulator on stack) instruction is executed, data is put on the stack and the pointer moved downward accordingly. When an RTS or PLA (PuLl Accumulator from stack) is executed, the corresponding data is pulled off the stack and the pointer moved back up. Since the stack pointer (which is a special location built into the main part of the computer) is just one byte, if you try and move it below location 256 it will wrap back around to location 511 and vice versa.

PAGES TWO THROUGH FOUR

Locations 512 to 1151, as you will see, are used by the OS as a workspace. Some are used for variables, some for tables, some for vectors, some for buffers, and some just for miscellaneous stuff. Now, a few words on using these locations. Don't, unless the description says you can! A lot of them are very important to the OS, and if you mess with them they may cause the computer to crash, which you don't want to happen. Keep in mind, though, that no matter what you do you can't hurt the computer (unless you throw it at a wall in frustration). You'll just hurt your program.

Also, be careful of locations that don't appear to be used. Atari has warned that these locations may be used in future versions of the OS, so stay away if you want to make sure your programs will work on all machines.

Let's jump right into page two. The first 42 bytes are used for interrupt vectors, so we'd better take a quick look at interrupts. As you remember, we first saw interrupts at location POKMSK (16). If you don't remember, go back and re-read that section. I'll wait for you here.

Back again? OK, so now we have the basic idea of what an interrupt is. The type of interrupt we saw at POKMSK is called an Interrupt ReQuest (IRQ). There's another kind of interrupt called a Non-Maskable Interrupt (NMI). What's the difference? Well, there's an assembly language command called SEI (SEt Interrupt disable). It tells the 6502 (the main chip) to ignore IRQ-type interrupts. Unfortunately, it can't tell the 6502 to ignore the NMIs. They are taken care of by another chip, called ANTIC, and so ANTIC is where you must go if you want to ignore NMIs.

The NMIs consist of the Vertical Blank Interrupt (VBI), the Display List Interrupt (DLI), and the SYSTEM RESET interrupt. We'll be seeing the interrupt vectors for both IRQs and NMIs in the next few locations, along with how to use them. An interrupt vector tells the OS where to go when the corresponding interrupt occurs (assuming it hasn't been disabled).

You might also want to look at IRQEN (53774), NMIEN (54286), and NMIST (54287) for more information on interrupts.

VDSLST
512,513 0200,0201

This is the vector for the Display List Interrupt (DLI) which is an NMI as we discussed in the last location. DLIs interrupt the screen drawing process so you can do things like change the screen color halfway down. They exist entirely for your benefit, the OS doesn't use them at all.

To get a DLI going, there are a couple of things you have to do. First, and most important, you have to decide what you want the interrupt to do! Write the routine to do it, making sure it ends with an RTI (ReTurn from Interrupt) instruction. Next, decide which row on the screen you want it to occur at (it will actually occur at the end of this row). Go into the display list and set the leftmost bit (bit seven) of the instruction for that row. That tells the display list that there is to be a DLI on this row. Now tell the OS where the DLI routine is by setting VDSLST (low byte and high byte of the routine address). Finally, you have to enable the DLIs. Do this by setting NMIEN (54286) to 192.

Here's a quick example from BASIC, simply reversing the playfield colors halfway down the screen:

```
100 GRAPHICS 0
110 DLIST=PEEK(560)+PEEK(561)*256
120 POKE DLIST+16,130
130 FOR MEM=1536 TO 1553
140 READ INSTR
150 POKE MEM,INSTR
160 NEXT MEM
170 POKE 512,0:POKE 513,6:POKE 54286,192
180 LIST
190 DATA 72,173,198,2,141,10,212,141,23,208
200 DATA 173,197,2,141,24,208,104,64
```

Make sure that the DATA is correct before you run the program. If it isn't, the computer might lock up. Here's an assembly listing of what those DATA statements represent:

```
0600 48     PHA
0601 ADC602 LDA COLOR2
0604 8D0AD4 STA WSYNC
0607 8D17D0 STA COLPF1
060A ADC502 LDA COLOR1
060D 8D18D0 STA COLPF2
0610 68     PLA
0611 40     RTI
```

Now that you know the basics, let me tell you a few limitations. First of all, there is very little time available during a DLI before the next row starts to get drawn. Make your routine short. Second, because an interrupt often occurs while something else is going on (like your BASIC program running), you have to make sure that you restore the accumulator and the X and Y registers if you use them. Do this by pushing their values onto the stack before you use them, and then pulling the values back off before you RTI. Finally, as should be painfully obvious to you BASIC programmers by now, this is most definitely machine language country. It's not very difficult machine language, but it is machine language.

A few notes now for the machine language programmers. Change the hardware registers, not the shadow registers. The shadow registers are used to update the hardware registers during VBLANK, so changing them halfway down the screen won't have any effect until VBLANK kicks in.

If you're going to have more than one DLI, then each DLI routine will have to reload VDSLST to point to the next one. The last one will have to point back to the first one. Make sure in this case that you enable the DLIs during VBLANK, or else they may not execute in the right order.

Use WSYNC (54282) if you're changing screen colors. When any value is stored in WSYNC, the next command won't be executed until the TV has finished drawing the current scan line. If you don't use it, your colors will change in the middle of a line and will flicker back and forth. Try it and see for yourself (get rid of "141,10,212" in line 190 and change "1553" in line 130 to "1550").

One other problem with DLIs is that pressing a key on the keyboard can cause DLI colors to "jump" down a scan line (try it). The solution? Well, the easiest is just not to use the keyboard. For more complex ways around it, you should consult De Re Atari.

DLIs are extremely powerful. They can be used to change colors, to change character sets, even to change player/missile positions and the fine scrolling registers, so be creative. Proper use of DLIs can produce a program that will do things you never thought the Atari was capable of.

VPRCED
514,515 0202,0203

This one's an IRQ vector, for an interrupt called the "serial proceed line interrupt," where the word "serial" indicates I/O to a peripheral such as the disk drive. It is initialized to 59314, which just holds a PLA and an RTI (i.e., the interrupt is not used).

VINTER
516,517 0204,0205

Another IRQ, this time for the "serial bus I/O interrupt." Initialized to 59314 again because it isn't normally used. Both VINTER and VPRCED's interrupts are processed by the PIA (Peripheral Interface Adapter) chip.

VBREAK
518,519 0206,0207

IRQ again, for the machine language BRK command (which is not the same as the BREAK key; see POKMSK [16] and BRKKEY [17]). It's also initialized to 59314.

VKEYBD
520,521 0208,0209

From now on, if I don't tell you what kind of interrupt it is, it's an IRQ, OK? There's a whole bunch of these suckers and only so many ways to say "here's another IRQ."

So here's another IRQ. This one occurs whenever a key other than BREAK is pressed (START, OPTION, and SELECT don't count because they're buttons, not keys). It's initialized to 65470 which is the OS keyboard IRQ routine (it makes sure that only one character gets printed when you press a key, and resets ATRACT [77]). If you want to put your own routine in, this is the place to do it. Keep in mind, however, that your routine will be executed before the key code gets converted to ATASCII (see the OS manual for a list of key codes).

The following three vectors are used to control communication between the serial bus and the serial bus devices (serial refers to the fact that bits are sent or received one after the other in succession). A much simplified explanation of this process follows. You should consult De Re Atari if you need more details.

The data being sent or received is stored in a buffer. If we're doing output, then a byte gets transferred from the buffer over to the serial output register (an interrupt routine does this). SIO takes it from there and puts it in POKEY's serial output shift register. POKEY then picks it up and sends it out one bit at a time. An interrupt is then generated and the whole process starts over. This goes on until the checksum byte has been sent, at which time a "transmit done" interrupt is generated and SIO hands control back to the main program, which has been waiting patiently all this time.

The process is pretty much the same if we're receiving data, except in reverse.

VSERIN
522,523 020A,020B

This is a good one. The "POKEY serial I/O bus receive data ready" interrupt vector. It means that this vector is used when the I/O bus indicates that it has received a byte that is now waiting in the serial input register, ready to be moved to a buffer. The routine in the OS to do this is at 60177, and that's what VSERIN is initialized to.

VSERIN is also called INTRVEC by DOS, which changes its value to 6691, a routine in DOS that does pretty much the same thing as the one in the OS, except in a different place.

VSEROR
524,525 020C,020D

The opposite of VSERIN, VSEROR is used when the I/O bus is ready to send a byte. Its official name is the "POKEY serial I/O bus transmit data ready" interrupt vector, which should make more sense this time. It is initialized to 60048, the address of an OS routine that, logically, moves the next byte in the buffer to the serial output register (from whence it gets sent). DOS messes with this one too, changing it to 6630, the address of its routine to do the same thing.

VSEROC
526,527 020E,020F

Another long-winded name: the "POKEY serial I/O bus transmit complete" interrupt vector. Since I'm sure you're all becoming experts at interpreting these names, it should come as no surprise that this vector is used when all the data has been sent. It is initialized to 60113, a routine that, when the checksum byte is sent (see CHKSUM [49]), sets the "transmission" done flag at XMTDON (58) and disables this kind of interrupt.

The following three locations are the interrupt vectors for the POKEY timers, all of which are initially unused and therefore set to the PLA/RTI combination at location 59314. The timer interrupt occurs when the associated timer counts down to zero.

For more information on the POKEY timers, see the section on timers right before location 53760.

VTIMR1
528,529 0210,0211

Interrupt vector for POKEY timer one (see AUDF1 [53760,53761]).

VTIMR2
530,531 0212,0213

Interrupt vector for POKEY timer two (see AUDF2 [53762,53763]).

VTIMR4
532,533 0214,0215

Interrupt vector for POKEY timer four (see AUDF4 [53766,53767]). This vector only exists in the "B" version of the OS.

VIMIRQ
534,535 0216,0217

Every IRQ vectors through this location on its way to the individual interrupt routines. It is initialized to 59126, the address of an OS routine that looks at IRQST (53774) to determine what kind of interrupt occurred and then jumps through the appropriate vector.

Attention B OS owners!

Since a lot of addresses in the new "B" version of the OS got shifted around, some of the initialization addresses given aren't the same in that version (which is now in a majority of the Atari's out there). Here are the changes (Figure 9).

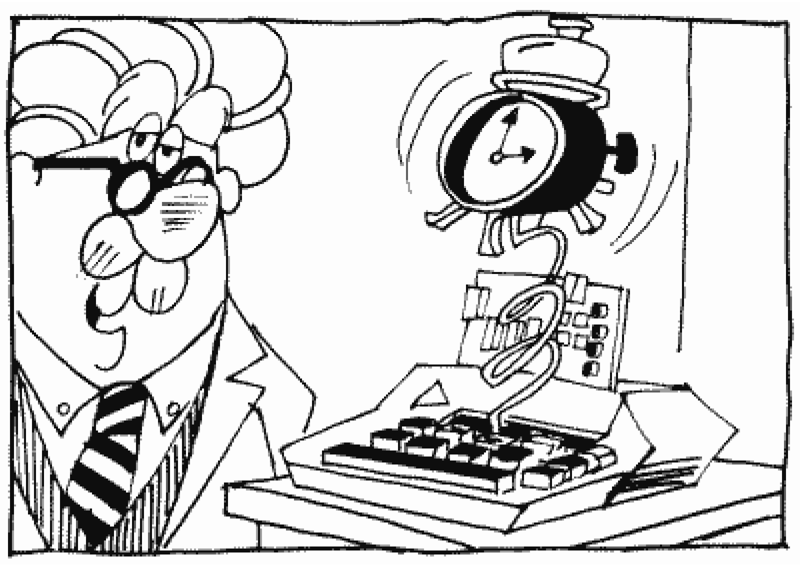
VECTOR	**INITIAL VALUE**
VDSLST	59280
VPRCED	59279
VINTER	59279
VBREAK	59279
VKEYBD	same as before
VSERIN	60175
VSEROR	same as before
VSEROC	60111
VTIMR1-4	59279
VIMIRO	59142
VVBLKI	59310
VVBLKD	59653

FIGURE 9. Vector list

Software timers

There are two types of timers in the Atari: software and hardware. We've already come across the hardware timers (see VTIMR1-4 [528-533]) and we're about to learn everything we never wanted to know about the software timers, which use locations 536 to 558. But first, a few words from our author.

There are, of course, differences between software and hardware timers, and you'll probably want to know them before you go running off into timer land. The biggest difference comes from the names.



Hardware timers are built into the POKEY chip, software timers are a part of RAM. The big difference comes in the way they keep time. You recall from location RTCLOK (18-20) that a jiffy is a sixtieth of a second, the amount of time it takes the television set to fill the screen. Well, the software timers count down by one every jiffy. The hardware timers, on the other hand, count down by an amount less than a jiffy, which you can specify (see locations 53760 through 53769). So, if you want to time things that take longer than a jiffy, use the software timers. Otherwise, go for the hardware.

CDTMV1
536,537 0218,0219

This is the first software timer (affectionately known as “system timer one”). Every VBLANK, the value in CDTMV1 gets decremented by one. When it reaches zero, a flag gets set so the OS knows to JSR through CDTMA1 (550,551). An important thing to note here is that the decrementing for this timer (and only this timer) is done during state one VBLANK. This means that CDTMV1 (along with RTCLOK [18-20] and ATRACT [77]) is updated every VBLANK, no matter what’s going on elsewhere in the computer. The rest of the software timers, on the other hand, are updated during stage two, which means that during time-critical I/O (like disk and cassette I/O; see CRITIC [66]), the other timers are not updated. Unfortunately, the OS knows this too, so it uses CDTMV1 for I/O routines. So, you see, we have a catch-22 situation here. Oh well! If you’re doing your own time critical routines though, you know which timer to use.

CDTMV2
538,539 021A,021B

This is system timer two, of course. When it reaches zero, it JSR’s through CDTMA2 (552,553). And, unless you slept through the last paragraph, you should already know that it will not be updated during time-critical I/O.

CDTMV3
540,541 021C,021D

The third system timer, again hampered by time-critical I/O. This one has problems of its own though. First of all, the cassette handler uses it. Secondly instead of JSRing through a vector when it gets down to zero, it just clears a flag at CDTMF3 (554). So don’t use it during cassette operations and don’t expect it to go anywhere after it’s done.

CDTMV4
542,543 021E,021F

Let’s see. You’ve already figured out that this is system timer four, that it doesn’t work during time-critical I/O, and you may have guessed that it clears a flag at CDTMF4 (556) when it’s done instead of vectoring. What’s left for me to say?

CDTMV5
544,545 0220,0221

The last of the timers. This one is no different than the last one except that the flag it clears is at CDTMF5 (558). But since you’re getting to know these things so well, I shouldn’t have to tell you that.

VVBLKI
546,547 0222,0223

Since this is the vector for the VBLANK interrupt (VBI), I suppose this is probably a good time to explain exactly what vertical blank is. With all the previous mentions of jiffies in this book, you should know by now that a jiffy is a sixtieth of a second. It is important because that's the time it takes the television set to fill the whole screen with a picture. Since the screen can't hold on to that picture for very long, the TV keeps drawing the picture over and over again, even if it doesn't change. It draws it one line at a time, from top to bottom. When it gets to the bottom, it stops drawing and goes back to the top, where it starts all over again. Now, the important part for us is when it stops drawing. At that time it tells the computer, "Hey, I'm not drawing to the screen anymore," thus generating a vertical blank interrupt. You should be able to see where the name comes from now. Incidentally, there is also a horizontal blank, which occurs while the TV has finished drawing one line and is on its way to the beginning of the next. Store any value in WSYNC (54282) and the computer won't do anything until the next HBLANK occurs.

Back to VBLANK. There are a few reasons why the TV isn't drawing to the screen. First of all, it gives us a way to time things, since VBLANK occurs precisely every sixtieth of a second. Secondly, nothing is being drawn to the screen during this time, so any graphics changes made during VBLANK will result in smooth, instantaneous changes on the screen. But, perhaps most importantly, VBI code runs independently of mainline code. What does that mean? It means that VBI code is essentially a separate program, running at the same time as your regular program! I wrote one VBI program, for example, that allowed the computer to play music at the same time I was typing in programs. Chris Crawford, in his classic Eastern Front 1941 game, used VBI to separate the thinking process of the game from the tedious stuff like graphics and user input. That allowed the computer to think about its next move at the same time the player was thinking about his or hers, thus simulating a true one-on-one situation. As you can see, VBLANK is an extremely powerful tool.

Let's take a closer look at what normally goes on during VBI. First of all, there are two stages. The first stage is always executed, while the second gets ignored if the time-critical I/O flag at CRITIC (66) is set. The first stage is called "immediate" vertical blank, the second is "deferred."

VVBLKI is the vector for the immediate stage so the OS goes through VVBLKI when the VBLANK interrupt first occurs. During this stage the real-time clock (RTCLOK [18-20]), the attract mode (ATRACT [77], DRKMSK [78], and COLRSH [79]), and system timer one (CDTMVI [536, 537]) get updated, processed, and so forth. Then CRITIC is checked. If it's set, indicating that the interrupt occurred in the middle of a time-critical I/O operation, the OS returns from the interrupt. If it's not, then it's OK to go on to stage two, so we do. When the OS is done with stage two (see the appendices for a complete list of what's done here), it vectors through VVBLKD (548,549) to the user's deferred VBI routine, and then finally returns from the interrupt when it's done there.

VVBLKI is initialized to point to SYSVBV (58463), which contains a JMP instruction to the OS stage one code (located at 59345 in the old OS, 59310 in the new one). If you change VVBLKI to point to your own routine, and you still want the OS code to be executed, you should end your routine with a JMP SYSVBV statement.
Whew, what a lot of mumbo jumbo! If you managed to plod through all of that, take a well-deserved rest. When you're done, we'll take a look at how you can use vertical blank for your own routines.

VVBLKD
548,549 0224,0225

Don't worry, there's still more to come on VBIs! This just seemed like a good time to formally introduce VVBLKD, the vector for the user's deferred VBI routine. The OS initializes VVBLKD to its "exit vertical blank" routine (at 59710 in the old OS, 59653 in the new one). If you use VVBLKD to point to your own routine, make sure to end that routine with a JMP XITVBL (XITVBL contains a JMP instruction to the exit vertical blank routine, which means you don't have to worry about which OS is being used since XITVBL is at 58466 in both). Note that you can also avoid the whole entire OS VBI code by writing your own immediate VBLANK routine and ending it with a JMP XITVBL instead of a JMP SYSVBV. Remember that none of the timers or color registers or anything will be updated if you do this (unless you update them in your routine).

By now you're probably either real excited over the prospect of using VBls yourself, or you're asleep. If it's the latter, then you're not even reading this because your eyes are closed, so I'm only going to deal with those of you who are excited, OK? Let's look at how to write our own VBLANK routines.

The first step is to decide whether you want your routine to be immediate or deferred. Most of the time it doesn't matter. There are, however, the following conditions which will require one over the other.

1. If you want to change locations that the OS deferred routine also changes (see Appendix Ten), you obviously want to do so after the OS does. Use deferred.

2. The maximum amount of time you can spend in immediate VBI is 2000 machine cycles (see a book on 6502 assembly language for information on the number of machine

cycles per instruction). If your routine is going to be long, you should therefore put it in deferred VBI, which has 20,000 cycles available. If you don't, things are going to look mighty funny on the screen. If you do use deferred, do your graphics first, since some of those 20,000 cycles occur while the screen is being drawn.

3. If you need your routine to be executed every VBLANK, regardless of whether time-critical I/O is occurring, use immediate. Be careful, however, that your routine will not cause problems with the I/O.

Now that you've decided what it should be (and you've presumably written it and put it in memory somewhere), all you need to do is change VVBLKI or VVBLKD to point to it. A simple task, right? Not quite. What happens if a VBI occurs while you're changing the vector? Crash city!

To make sure this doesn't happen, you have to change the vectors during VBLANK. But that itself presents a small problem. How do we get into VBLANK to change the vectors if we have to change the vectors to get to VBLANK (good old catch-22 again)? Luckily, Atari has thoughtfully provided a VBI routine that makes the change for you. It's called SETVBV and is at 58460. To use it, load the 6502 Y register (LDY) with the low byte of the address for your routine, and load the X-register (LDX) with the high byte. Then load the accumulator (LDA) with a six if you want immediate VBI, seven if you want deferred, and JSR SETVBV. Now your VBI will be up and running.

Here's a simple example that uses location CHACT (755) to make inverse text blink:

```
100 FOR MEM=1536 TO 1575
110 READ CODE
120 POKE MEM,CODE
130 NEXT MEM
140 X=USR(1536)
150 DATA 104,169,0,141,29,2,160,16,162,6,169,6,141,29,2,32
160 DATA 92,228,96,173,28,2,208,13,169,30,141,28,2,173
170 DATA 243,2,73,2,141,243,2,76,95,228
```

Make sure that the DATA values are correct before you run the program. If they aren't, the computer will probably crash and you'll lose the program.

Here's the assembly language listing of the machine code (which is stored in the DATA statements):

```
0600 68              PLA
0601    A900         LDA #$00
0603  8D1D02         STA CDTMV3 + 1
0606    A010         LDY #VBLANK&255
0608    A206         LDX #VBLANK/256
060A    A906         LDA #$06
060C  8D1D02         STA CDTMV3 + 1
060F  205CE4         JSR SETVBV
0612      60         RTS
0613  AD1C02 VBLANK LDA CDTMV3
0616    D00D         BNE VBLXIT
0618    A91E         LDA #$1E
061A  8D1C02         STA CDTMV3
061D  ADF302         LDA CHACT
0620    4902         EOR #$02
0622  8DF302         STA CHACT
0625  4C5FE4 VBLXIT JMP SYSVBV
```

The "LDA #$1E" in the preceding listing is used to specify a half second interval ($1E hex equals 30 decimal equals 30 jiffies equals half a second) for use in blinking. Make it larger or smaller to make the interval longer or shorter, respectively.

CDTMA1
550,551 0226,0227

CDTMA1 is the vector for system timer one (CDTMV1 [536,537]). It's initialized to 60400, which is the address of a routine to set the time out flag TIMFLG (791). This is because the OS uses CDTMV1 for I/O routines, which is a very good reason why you probably should use timer two instead.

The OS vectors through CDTMA1 when CDTMV1 counts down to zero. If you do use CDTMV1, and are setting it for a value greater than 255 (i.e., setting both the low and the high byte), this presents a potential problem. Since CDTMV1 is updated during VBLANK, and there is a chance that a VBLANK might occur while you're setting CDTMV1, you should set the low byte first. You can also use the SETVBV routine mentioned in the VBLANK description preceding. Just LDY with the low byte, LDX with the high, LDA with the timer number (1-5), and JSR SETVBV. This will assure that the timer gets set during VBLANK.

Since the OS JSRs through this vector, you should end your routine) with an RTS instruction.

Incidentally, CDTMV1 reaching zero generates an NMI, which then does the vector.

CDTMA2
552,553 0228,0229

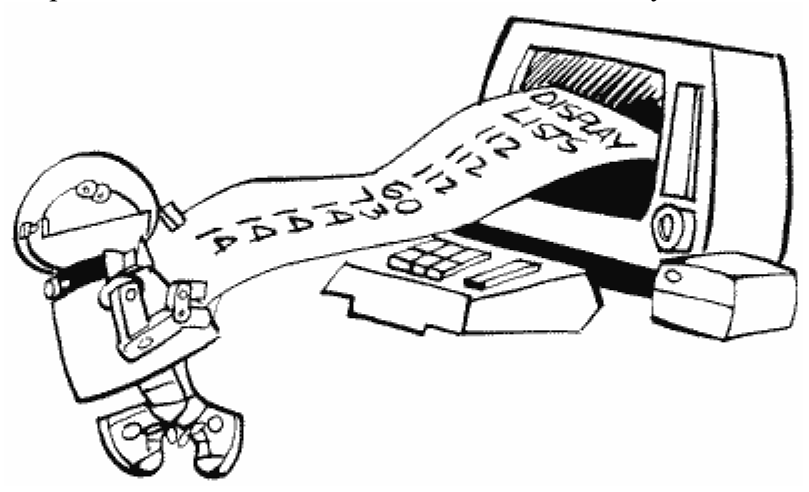
Same as CDTMA1, except this one is not used by the OS and is therefore initialized to zero. Oh, and of course CDTMV2 (538,539) reaching zero causes the vector through here, not CDTMV1. But then we already knew that, didn't we?

CDTMF3
554 022A

Unlike system timers one and two, timers three through five merely clear a flag when they count down to zero. This is the flag for CDTMV3 (540,541) and is also used by DOS as a timeout flag, so beware of possible conflicts if you use it.
As with the other two flags, you must set CDTMF3 when you set CDTMV3. Any nonzero value is ok.

SRTIMR
555 022B

Well, here in the middle of all the timer stuff is a different kind of timer. As everybody knows, if you hold down a key on the Atari, it will start repeating, right? And something has to tell the OS how long to wait before starting that repeat and before repeating it again, right? And can you guess what location does that? Sure, I knew you could. SRTIMR is set to 48 every time a key is pressed. Every stage two VBLANK that the key is still held down, SRTIMR gets decremented by one. When it reaches zero, the repeat process starts. It gets set to six, decremented again, the key repeats, it gets reset to six, and so forth until the key is released. Unfortunately, there are no locations that store the two delay times, so you can't speed up or slow down the process just by changing a couple of locations. There is, however, another way to do it.



As you recall, the initial delay time of 48 is set whenever a key is pressed. As you may or may not recall, we came across a vector a few locations ago (VKEYBD [520,521]) that pointed to the IRQ routine for a key being pressed. It is in this routine that the delay is set. So, in order to change the delay, you must essentially take the OS routine, change the delay value, store your revised version in memory, and update the vector. You'll find the OS routine at location $FFBE on page 130 of the OS listing.

How about the other delay, the six jiffy one once the repeat is started? If you were paying attention (and I know you were), you already know that it gets set in stage two VBLANK. Can you guess what you're going to have to do to be able to set it yourself? If you guessed "take the OS stage two VBLANK interrupt routine and put it in my own deferred VBI routine with the delay value changed," then give yourself a pat on the back.

"But wait! The OS stage two VBI routine gets executed whether I have my own deferred VBI routine or not," you say, taking me completely by surprise. You're right though (or would have been if you had said it). Your deferred routine, however, happens after the OS's, so you can just repeat the part that sets the delay and, since you'll set it after the OS does, yours will be the one that counts. The part you want is at locations $E87C through $E897 on page 36 of the OS listing, and locations $E8E8 through $E8EE on page 37 (these locations will be different in the new OS, but that's irrelevant here). Be aware that the OS will now be executing this routine twice, and will therefore be decrementing by two every VBLANK. You should set SRTIMR to double the delay you want, and also change your deferred routine so that it resets SRTIMR if it's equal to zero or one. That makes sure that the OS routine doesn't reset it before you get a chance to.

CDTMF4
556 022C

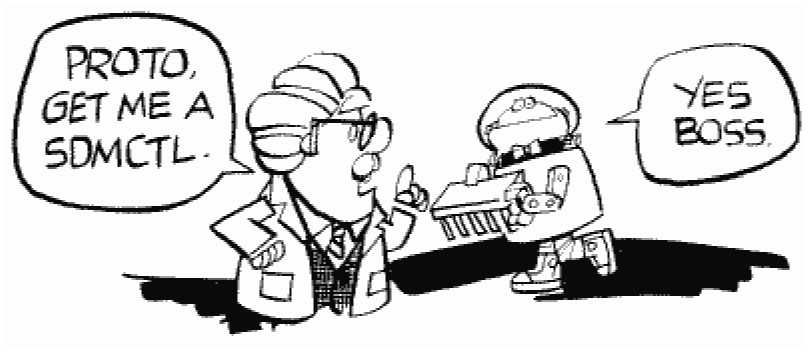
And now, back to our timers. This is the flag for CDTMV4 (542, 543). See CDTMF3 for more information.

INTEMP
557 022D

INTEMP is used for temporary storage during the SETVBL routine. As you recall, SETVBL is at the address stored in 58460. Heaven only knows what INTEMP is doing here in the middle of the system timers.

CDTMF5
558 022E

This is the flag for CDTMV5 (558,559). See CDTMF4 for more information (ha ha).



SDMCTL
559 022F

This location is amazing. So many things can be done here that you'll just flip! Maybe not. Anyway, SDMCTL controls something called Direct Memory Access (DMA). Simply put, DMA is the process by which ANTIC, the Atari graphics chip, gets the information from memory it needs to fill in the screen (this means information for the playfield and for player/missile graphics). Now this process obviously takes time and slows down the 6502 because of that. So what happens if we turn DMA off? Things will run faster. Try it; POKE 559,0 to turn DMA off. Uh-oh, what happened? The screen went blank. But of course, with no DMA, ANTIC isn't getting the information for the screen so there's no picture. What good does a computer do without a picture? Well, sometimes you don't need one. For example, if you're doing a lot of calculations, it's more important to get them done quickly than to have an "I'M THINKING" message on the screen, and turning off DMA will speed things up by as much as 30 percent! Of course, with a blank screen the user may think that the computer just up and croaked on him, so be sure you give a warning before the lights go off. SDMCTL is a shadow register for DMACTL (54272).

By the way, in case you're still sitting there after the POKE 559,0 with a lifeless computer, just press SYSTEM RESET or type in POKE 559,34. You can't see the POKE written on the screen, but it will still work when you press return.

OK, so we can turn off the screen. Big deal, right? Right, but it's what we can do when we turn the screen back on that counts. SDMCTL let's you make the playfield (the blue screen you PLOT and PRINT onto) wider, narrower, or nonexistent. It also lets you turn players and missiles on and off, define how tall you want the players to be, and, of course, turn ANTIC on and off.

Let's take a look at a breakdown of SDMCTL and see what does what (Figure 10).

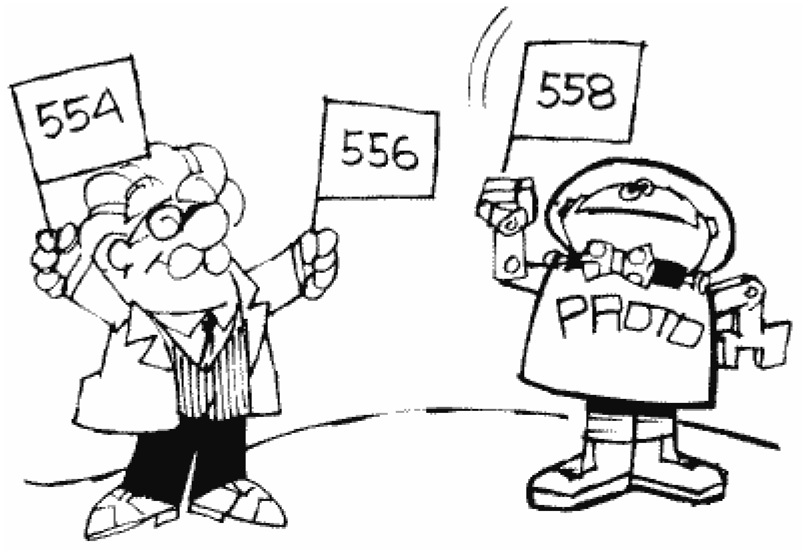
To get players and missiles going, see GRACTL [53277] as well.

BIT(S)	PATTERN	VALUE	RESULT
0,1	------00	0	No playfield
	------01	1	Narrow playfield
	------10	2	Regular playfield
	------11	3	Wide playfield
2	-----0--	0	Missiles off
	-----1--	4	Missiles on
3	----0---	0	Players off
	----1---	8	Players on
4	---0----	0	Double height players
	---1----	16	Regular height players
5	--0-----	0	ANTIC off
	--1-----	32	ANTIC on
6,7	--------		Not used

FIGURE 10. SDMCTL chart

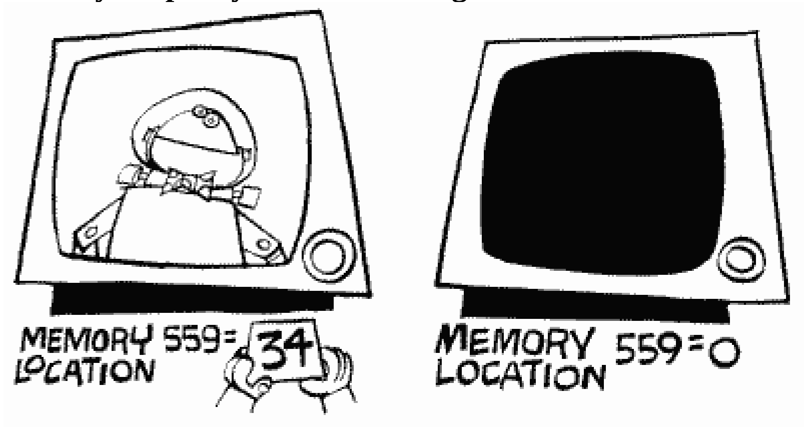
So, to use SDMCTL, pick the options you want, add their values, and POKE away. Note that ANTIC must be turned on if you want a display, and you can only have one type of playfield at a time. While we're on the subject of playfields, you'll probably want to know exactly what a "narrow" and "wide" playfield are. OK, here goes:

The width of the playfield is measured by something called a "color clock." A color clock is twice as wide as a pixel in graphics mode eight. That means that a character in graphics mode zero is 4 color clocks wide (we'll use graphics mode zero as an example since you can see the playfield in this mode), and that in turn means that the regular playfield is 160 color clocks wide (forty characters times 4 color clocks per character). A narrow playfield is only 128 color clocks (32 characters) wide, while a wide playfield has 192 color clocks (48 characters). The television set draws 228 color clocks total (including the black border), but not all of these can be seen. As a matter of fact, not all of the 192 in a wide playfield can be seen either, which makes it good for horizontal scrolling.



Is having a wide or narrow playfield as easy as it sounds? Well, yes and no. Getting it on the screen's easy (try POKE 559,35 right now), using it properly isn't. Unfortunately, telling SDMCTL that you want a different size playfield doesn't tell the OS that anything's different. To see what I mean, try POKE 559,35 from graphics mode zero. Now you have 48 characters per line, but the OS still thinks you have 40. Try typing stuff in and you'll see the problem. There is no way around this problem, which means that you have to set up the screen memory yourself if you want to take advantage of this feature. Sorry.

Double-height players, in case you were wondering, have dots the height of those in graphics mode seven, while regular-height players are the height of graphics mode eight. Despite the way I named the two, double-height players are given to you unless you specify otherwise using SDMCTL.



SDLSTL
560,561 0230,0231

Another important location. SDLSTL holds the address of the display list. Let's talk display lists.

We already know about screen memory, the memory locations that hold the information that is to be displayed on the screen (see SAVMSC at 88,89). How does the computer know how to interpret this memory, though? As we learned in SDMCTL there is a special chip called ANTIC that takes care of the graphics. ANTIC has a list of commands that tells it how to display the screen memory. Oddly enough, this list is called the "display list." Since the display list is made up of commands, it's actually like a little program. And, since the screen memory has to be redisplayed 60 times a second, this program is a continuous loop, running over and over again. Why does the screen memory have to be redisplayed? The TV set draws a picture by making different parts of the screen glow at different brightnesses. The screen, however, will only glow for a very short period of time. Therefore, in order to get a picture to stay on the screen, the TV has to draw it 60 times a second.

For now, let's pretend that a display list is written just like a BASIC program, only with special commands. Let's look at what such a display list would look like:

```
100 DRAW 8 BLANK SCAN LINES
110 DRAW 8 BLANK SCAN LINES
120 DRAW 8 BLANK SCAN LINES
130 CHARACTER MODE 0 LINE...
140 ...WITH SCREEN MEMORY STARTING AT <address>
150 CHARACTER MODE 0 LINE
160 CHARACTER MODE 0 LINE
.
.
.
370 CHARACTER MODE 0 LINE
380 GOTO 100 AND WAIT FOR VBLANK
```

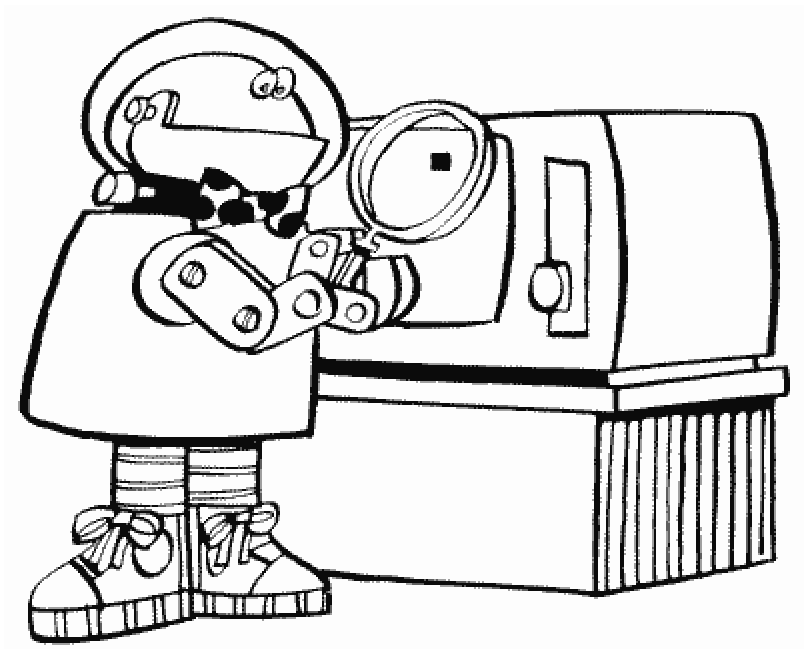
We start by telling ANTIC to leave the first 24 scan lines blank. Scan lines are the height of a graphics mode eight line, start before the left edge of the screen and go all the way past the right edge. If you look closely at the screen you can even see them. We leave 24 lines blank so that we can be sure that all of our picture will be on everyone's screen. TV and monitor screens all act slightly differently, so the blank lines will create a frame that can cover the edges of the screen. These blank lines make up the top of the black border that you can see in graphics mode zero.

Next we want to start our mode zero lines, so we have a mode zero line command. ANTIC has to know where the screen memory is before it can start drawing, so we make the first mode zero command a special one that tells ANTIC that the address

of the screen memory will come next. Then, after the screen memory address, we have 23 regular mode zero commands. Finally, we tell ANTIC to go back to the beginning and start all over again after VBLANK. Remember that VBLANK is the time during which the TV is getting ready to start drawing the picture again. We want to make sure it's ready before ANTIC starts again, we so tell ANTIC to wait until VBLANK is over.

Now that we have a basic understanding, let's look at the specifics. First of all, ANTIC doesn't use line numbers. In a real display list, the line numbers would be memory locations. Secondly, ANTIC has abbreviations for all the commands. And thirdly, there is no thirdly. Let's therefore look at the proper way to write the preceding display list (we'll have it start at location 1000 [decimal], although it would usually be much higher in memory):

```
1000 BLK 8
1001 BLK 8
1002 BLK 8
1003 CHR 2 LMS
1004 < screen memory low byte >
1005 < screen memory high byte >
1006 CHR 2
1007 CHR 2
.
.
.
1028 CHR 2
1029 JVB 1000
```



As you can see, this isn't that much different from our original. LMS, by the way, stands for Load Memory Scan and tells ANTIC that the next two bytes will be the address of the beginning of screen memory. Now, the final step is to convert these commands into numbers that we can POKE into memory. There is a unique number assigned to each command, and the chart in Appendix Twelve gives you those numbers. Before you look at that chart, however, let me explain the other commands that you'll see there:

MAP is the same as CHR, except it's used to indicate a graphics mode rather than a character (text) mode.

JMP is like JVB, except it doesn't tell ANTIC to wait for the end of VBLANK. It's needed because of a quirk in ANTIC that says a display list can't cross a 1K boundary. What's a 1K boundary? It's a memory location that's a multiple of 1024. With display lists created by GRAPHICS commands, this is no problem. If you're designing your own, however, and you have to cross such a boundary, JMP over it. While we're on the topic of boundaries, you should also be aware that screen memory is not allowed to cross a 4K boundary. If it does, you have to have a second LMS instruction to get past the boundary. Under normal circumstances, however, this only happens in graphics modes 8 through 11, and the OS will take care of it for you.

HSC, like LMS, is not really a command but rather a modification to a command. It tells ANTIC that this mode line is to have the capability of fine horizontal scrolling (see HSCROL at 54276).

VSC, you guessed it, is another modification that specifies a fine vertical scrolling capability. See VSCROL (54277).

DLI is the fourth modification, telling ANTIC that there is to be a display list interrupt at the end of this mode line. Pay particular attention to the "end." See VDSLST (512,513) for more details on DLIs.

Now that you'll be able to understand the chart, why don't you go take a quick peek at it.

Run the following program to take a look at the actual graphics mode zero display list as it is stored in memory:

```
100 GRAPHICS 0
110 DLIST=PEEK(560)+PEEK(561)*256
120 PRINT PEEK(DLIST)
130 IF PEEK(DLIST)<>65 THEN DLIST=DLIST+1:GOTO 120
140 PRINT PEEK(DLIST+1)
150 PRINT PEEK(DLIST+2)
160 END
```

Use CTRL-1 to pause the display list. Notice that the last two numbers PRINTed (the address for the JVB) are the same as the values in 560 and 561. If you can't figure out why, drink a couple of cups of coffee and read this whole description all over again. Here is a hint: They tell the computer to go back and use the same display list all over again. If you change the numbers here the computer will use another display list at the new address, which means you could use several display lists at once.

We've pretty much covered the standard GRAPHICS display lists, but what about custom ones? It should have occurred to you by now that you can write your own display lists, mixing different graphics modes on the screen. Since this is true, and is such a popular thing to do, there is a special appendix at the end of the book telling you just how to do it. So please consult it if you want to write your own display list.

A few (more) words before we move on. What can you use LMSs for other than to tell ANTIC where the screen memory is? Nothing! You can, however, tell ANTIC where the screen memory is more than once in the same display list. Why would you want to do that? Well, you have to if you're fine scrolling (see HSCROL and VSCROL). You could also do it to repeat the same line over and over again without wasting memory. LMSs are another powerful tool that have no steadfast rules about what to use them for; use your creativity. Here's an example of repeating text. After the program has run, clear the screen and try typing in a line of text.

```
100 GRAPHICS 0
110 DLIST=PEEK(561)*256+67:POKE 560,67
120 LOW=PEEK(88)
130 HIGH=PEEK(89)+2
136 POKE 89,HIGH
137 POKE DLIST,112
138 POKE DLIST+1,112
139 POKE DLIST+2,112
140 DLIST=DLIST+3
150 FOR ROW=0 TO 23
160 POKE DLIST,66
170 POKE DLIST+1,LOW
180 POKE DLIST+2,HIGH
190 DLIST=DLIST+3
200 NEXT ROW
210 POKE DLIST,65
220 POKE DLIST+1,PEEK(560)
230 POKE DLIST+2,PEEK(561)
```

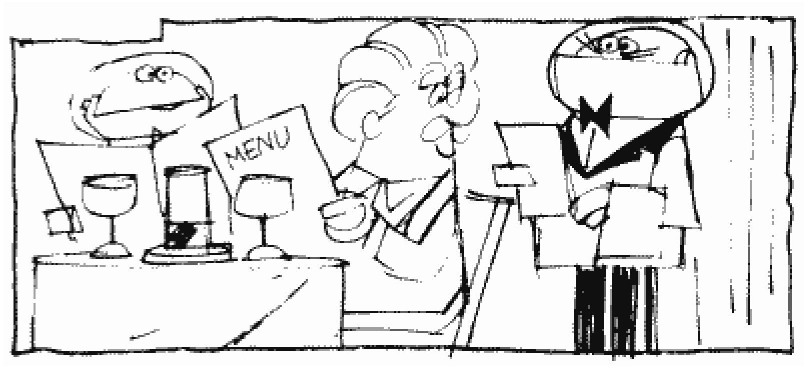
What's going on here? Essentially we're rewriting a graphics mode zero display list so that all the lines have LMSs that point to the same address. We also have to change SAVMSC (88,89) so that the OS knows where our new screen memory is. Why isn't the new screen memory in the same place as the old screen memory?

Because our new display list overflows into the old screen memory, that's why.

If you press SYSTEM RESET a normal graphics mode zero screen will appear.

SSKCTL
562 0232

SSKCTL is used to control the serial port, and is a shadow register for SKCTL (53775). As your state of confusion should indicate to you, it is not really a location for the inexperienced. Look at SKCTL if you are interested, look at the OS manual if you're really interested



SPARE
563 0233

This location is currently unused. Atari reserves the right to use it in future versions of the OS, so don't count on it being safe to use.

LPENH
564 0234

Ever hear of a light pen? The thing that looks like a pen and you can draw on the screen with it and so forth? Well, if you happen to be one of the lucky few who have one, this will tell you what horizontal position on the screen it's pointing to. Neat, huh?

LPENH is a shadow register for PENH (54284).

LPENV
565 0235

This is the vertical position of the light pen on the screen. It's a shadow register for PENV (54285).

Since light pens defy all reason, a few words about them are probably in order here. Firstly, LPENH and LPENV are set when the light pen is pressed to the screen.

LPENV gets the value of VCOUNT (54283) when the pen was pressed (VCOUNT gets incremented by one every two scan lines). LPENH, on the other hand, gets a value based on the number of color clocks that have been drawn so far (see SDLSTL for an explanation of a "color clock"). Now I'd be more than happy to give you the range of values that LPENH and LPENV will return as you move a light pen about the screen, but unfortunately the values depend on several things. Run the following program to see what the limits are for your computer. Also see STICK0-3 (632-635).

```
100 GRAPHICS 0
105 POKE 752,1
110 POSITION 2,11
120 PRINT "Light pen horizontal position = "
130 PRINT "          vertical position = "
140 POSITION 34,11
150 PRINT PEEK(564);" ";
160 POSITION 32,12
170 PRINT PEEK(565);" ";
180 GOTO 140
```

You should also note that light pens are not very precise; the values can vary slightly even if you hold it steadily at one point on the screen. If you're writing a program that uses the light pen, be sure to allow for a little variation in the values. This can be done by "sampling" values. Basically this means read 10 or so values every time you need one, throw out the highest and lowest, and average the rest to get a single value.

BRKKY
566,567 0236,0237

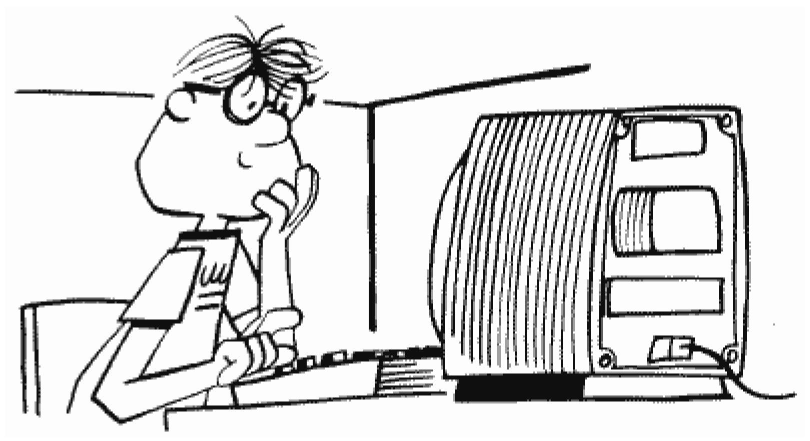
In the new "B" version of the OS, this is the vector for the BREAK key interrupt. It is initialized to 59220, which means you will find 84 in location 566 and 231 in location 567. To disable the break key, POKE 143 into 566.

See BRKKEY at location 17 if you want to write your own BREAK key routine and you have the old OS. Also see locations 512 through 535 for more information on interrupt vectors.

Noname
568,569 0238,0239

More "unused" bytes. See the warning at location 563 about such bytes.

OK, we're going back to I/O now. The next four bytes make up the "Command Frame Buffer" (CFB), a table that SIO uses when doing serial bus operations (remember that the serial bus is what information travels back and forth on). It is not designed to be used by you, so you should be reading out of curiosity rather than necessity. For more information on the CFB and the rest of the I/O system, make sure you read Appendix Seven.



CDEVIC
570 023A

CDEVIC contains the device code

CCOMND
571 023B

CCOMND contains the command code.

CAUX1
572 023C

CAUX1 contains the command auxiliary byte one, which comes from DAUX1 at location 778.

CAUX2
573 023D

Finally, CAUX2 contains the command auxiliary byte two, which SIO gets from DAUX2 at location 779.

TEMP
574 023E

SIO uses TEMP as a TEMPorary storage place (the people who name these things are so clever).

ERRFLG
575 023F

If any error occurred during device I/O, with the exception of a timeout, ERRFLG is set to 255. Otherwise, if everything is okey-dokey, it is set to zero.

Also see STATUS at location 48.

DFLAGS
576   0240

When a disk is booted (the computer is turned on with the disk drive on), the first disk "record" (a record is a segment of information that has been recorded on the disk) is read from sector one (a sector is a segment of the disk, shaped like a piece of pie) into memory. Some of the information from this record is used to continue the boot. The first byte in the record contains several useful flags. It gets stored in DFLAGS.

DBSECT
577   0241

The second byte tells how many sectors are used in the boot file. It gets stored in DBSECT.

BOOTAD
578,579   0242,0243

Finally, because we have to know where to put the file, the third and fourth bytes give the starting address. They get stored in BOOTAD. Once the OS knows BOOTAD, it moves the record it just read in over to the new address and starts loading in the rest of the file, putting each record after the previous one until the whole file is properly in memory.

BOOTAD gets transferred to RAMLO (4,5) which, because it is in page zero, is used to move the file from the sector buffer to its place in memory.
In most cases, the boot file is DOS, and BOOTAD will hold the address 1792.

COLDST
580   0244

COLDST is fun. The OS sets it to one during the power up process and then sets it to zero when everything has been properly initialized. If somebody presses SYSTEM RESET, the OS looks at COLDST to see whether it was in the middle of power up when SYSTEM RESET was pressed. If it was (COLDST equals one), then the turkey who did the pressing messed up the power up and the OS has to start all over again. Otherwise, the OS just treats it like a normal RESET.

Fun? That's your idea of fun? Hey, let me finish. The OS isn't too smart; COLDST is the only way it knows whether or not it's in the middle of power up when SYSTEM RESET is pressed. That means you can set COLDST to one in your program, and if SYSTEM RESET gets pressed, the computer will act as if somebody just turned the computer on. Your whole program will be erased rather than broken into (usually SYSTEM RESET will cause the OS to jump to BASIC,

where your program can be LISTed or SAVEd).

Use COLDST along with POKMSK (16) and STMCUR (138,139) to totally protect your BASIC programs from being looked at or SAVEd (the disk or cassette they're on could still be copied though). Another good use for this trick is to POKE 580,1 and press SYSTEM RESET instead of using your ON-OFF switch when you want to load in another program. It saves wear and tear on the computer.

No name
581 0245

Yet another unused byte for which the warning at location 563 applies.

DSKTIM
582 0246

The "disk timeout register." We last saw timeouts at location PTIMOT (28). Well, they're back, this time for the disk drive (PTIMOT was for the printer). DSKTIM holds the timeout value for the FORMAT command. It is supposedly initialized to 160, but I have seen machines that initialize it to 120 and 224, which I suspect has something to do with different versions of the OS. Anyway, regardless of what it's initialized to, the value in DSKTIM is updated after every STATUS request with the value in DVSTAT+2 (748).

You should look at DTIMLO (774) for lots more information on disk timeout, including the exact use for DSKTIM.

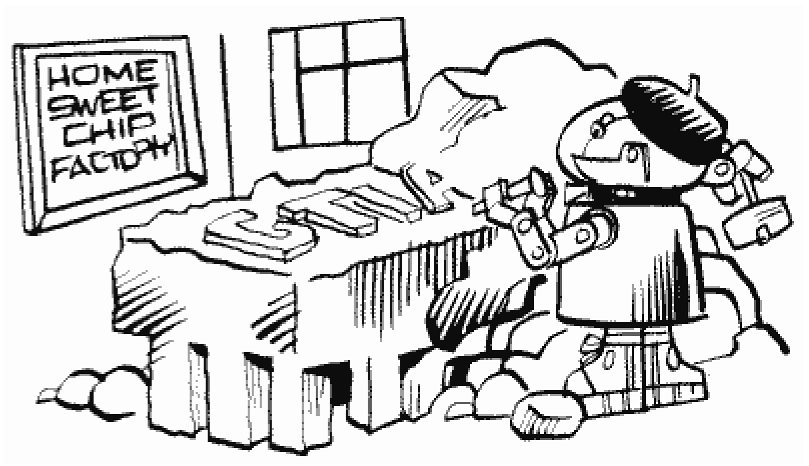
LINBUF
583-622 0247-026E

Hey, remember buffers? This is a 40-byte-long buffer used by the screen editor. You see, the screen editor needs a place to temporarily store a line of text when it's moving stuff around on the screen. This is it.

ADRESS (100,101) is used as a temporary zero page pointer to LINBUF during the moving process.

GPRIOR
623 026F

GPRIOR is used to set priorities and to select GTIA modes. Whoa boy, what in tarnation are priorities? I'm so glad you asked! There is a complete example of all this player missile stuff on the disk or tape we offered with this book. It will not only show you what all of this is, but since we never protect our programs, you can change many of the locations that our program uses for practice.



In the wonderful world of Atari graphics, there are two kinds of things that can appear on the screen. First of all, there is the playfield. The playfield is what you get by PRINTing, PLOTting, and DRAWTOing. The playfield is made up of as many as five colors, which are specified by the color registers (as in SETCOLOR color register, color, brightness). Each of these colors represents a different part of the playfield. The one with the same color as color register zero is called playfield zero and so forth. The one with the background color (color register four) is called BAK.

The second type of thing that can appear on the screen is player/missile graphics. We'll be getting more into players and missiles in the GTIA/GTIA chip at location 53248, but for now just be aware that there are four players and four missiles that can appear on the screen at the same time as the playfield.

So where is all of this getting us? Well, if you have player/missile graphics on the screen, and you also have a playfield, which should be seen when the two are in the same place? In other words, which should have priority over the other? GPRIOR tells ANTIC (the chip that draws the picture) who has the highest priority (i.e., who gets to be seen). Let's look at the chart in Figure 11.

GPRIOR	**…0001**	**…0010**	**…0100**	**…1000**	**BIT PATTERN**
	(0)	**(2)**	**(4)**	**(8)**	**(VALUE)**
HIGHER	P0	P0	PF0	PF0	
PRIORITY	P1	P1	PF1	PF1	
	P2	PF0	PF2	P0	
	P3	PF1	PF3+P4	P1	
	PF0	PF2	P0	P2	
	PF1	PF3+P4	P1	P3	
	PF2	P2	P2	PF2	
LOWER	PF3+P4	P3	P3	PF3+P4	
PRIORITY	BAK	BAK	BAK	BAK	

FIGURE 11. GPRIOR chart

PF means PlayField, P means Player. Missiles have the same priority as the player with the same number. Keep in mind that something with a higher priority will appear to move in front of something with a lower priority. Similarly, of course, something with a lower priority will appear to move behind something with a higher priority.

As you probably noticed, only the last four bits of GPRIOR are used to set the priority (make sure only one of those four bits is on). What about the other four? If you set bit four (---1---- [16]), then all the missiles will have the same color as playfield three. That lets you move the missiles together and use them as a fifth player (P4). If you set the bit five (--1----- [32]), then overlapping player zero and player one will produce a third color in the overlap area. This goes for player two and player three as well. For machine languagers, or just the curious, the third color is produced by ORing the colors of the two players together.

As long as we're on the subject of overlap colors, if you do set more than one of the last four bits, then in a case where two overlapping objects have the same priority, the overlap area will be black.

In graphics modes zero and eight, only the color of the text or pixels will be changed if a player or missile flies over them. The brightness will not change.

So now we're left with the seventh and eighth bits. They don't have anything to do with priorities or player missile graphics. Instead they indicate whether or not a GTIA mode is being used, and if so, which one. They work as shown in Figure 12.

00--------	(0)	No GTIA mode
01--------	(64)	GRAPHICS 9
10--------	(128)	GRAPHICS 10
11--------	(192)	GRAPHICS 11

FIGURE 12. GTIA chart

You only need to set these bits if you're writing your own display list. Otherwise the OS will take care of them when you use BASIC to call "GR. 9, 10, or 11"

If you want more information on GTIA, what it is, whether you have it, and how to use it, consult Appendix Four in the back of this book.

GPRIOR is a shadow register for PRIOR (53275).

The next 24 locations (624 through 647) hold information about the joysticks, paddles, and light pens.

PADDL0
624 0270

PADDL0 holds the current value of paddle zero (the left paddle in the leftmost plug in the front of the computer). Paddles can have a value ranging from 0 to 228; the further you turn the paddle clockwise, the higher the value.

Paddles are actually little more than a "potentiometer." Let's look at a potentiometer as though it were a bathroom faucet. The computer sends a value 255 worth of water into the faucet, but the amount that comes out depends on how far open the faucet is. If it's all the way open (the paddle is turned clockwise as far as it will go), then almost all of the water will flow through (228 worth). If it's all the way closed (the paddle is turned counterclockwise as far as it will go), then none of the value will flow through. And that's exactly how a potentiometer works, except the computer is sending electricity into it rather than water (paddles aren't water-

proof).

If you design a program that uses the paddles, be careful. Most of the time you won't want them to have a range of 0 to 228. For example, if you're using the paddles to move a player, the player will probably move off the screen. So what can you do to get the range that you want? Let's suppose you want to go from LOW to HIGH. First, do the following:

```
RANGE = HIGH-LOW + 1
EACH = RANGE/228
```

These two lines should come somewhere in your program before you start using the paddles. Then, every time you read the paddle, do the following:

```
MYVAL=INT(OLDVAL*EACH+.5)+LOW
```

where OLDVAL is the value of the paddle, and MYVAL is the corresponding number in the range you wanted.

PADDL0 is a shadow register for POT0 at location 53760. Note that the value for the button (trigger) on the paddle can be found at PTRIG0 (636).

The following paddle locations work the same as paddle 0, but you change the number of the paddle (of course). Paddles 0 and 1 plug into the leftmost port (hole) numbered 1 on the front of the Atari computer. Likewise 6 and 7 plug into the rightmost port numbered 4 on the computer.

PADDL1
625   0271

The value for paddle one and a shadow register for POT1 at 53761.

PADDL2
626   0272

The value for paddle two and a shadow register for POT2 at 53762.

PADDL3
627   0273

I'll leave this and the next four for you to figure out (hint: see the last three locations).

PADDL4
628 0274

PADDL5
629 0275

PADDL6
630 0276

PADDL7
631 0277

STICK0
632 0278

Let's see. PADDL0 was paddle zero, so I wonder what STICK0 is? Could it possibly be joystick zero? Despite all the odds against it, it is. Joystick zero is the one plugged into the leftmost plug in the front of the computer.

Unlike paddles values, joystick values don't appear at first (or second) to make much sense. Let's take a look at those values (Figure 13).

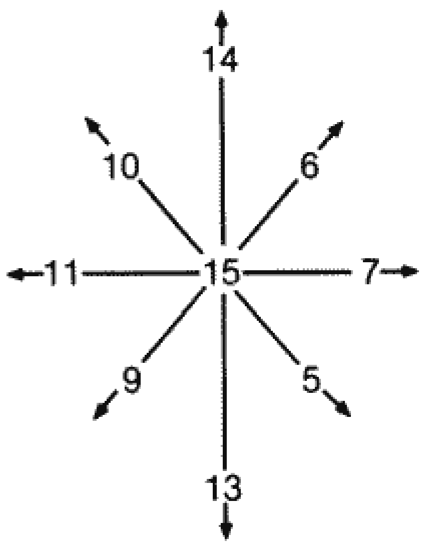


FIGURE 13. Joystick values in decimal

This figure represents the nine possible positions the joystick can be in, along with the value corresponding to each. If you move joystick zero up, for example, STICK0 will have a value of 14. Now, unless you have some brilliant power of observation that I don't, these values don't seem to make any sense. I mean, does "14" mean "up" to you? Not to me. They must make some kind of sense to the computer, however, so let's take a look at them again (Figure 14), this time in binary (the way the computer sees them).

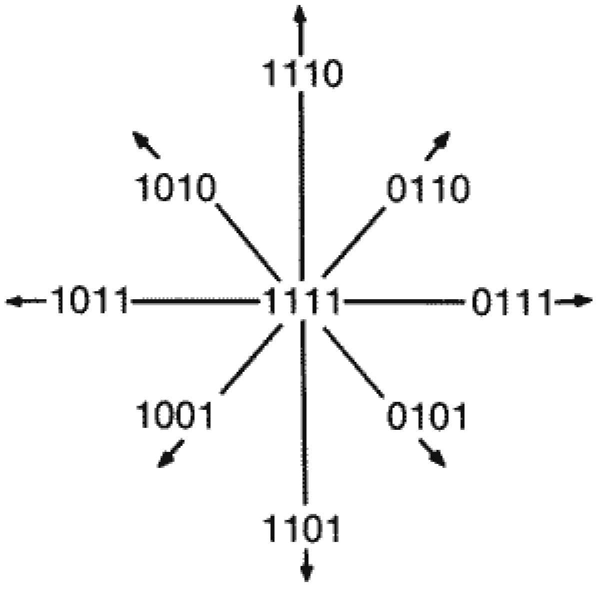


FIGURE 14. Joystick values in binary

It may not be immediately obvious, but now things make sense. Notice how the first bit (the digit on the right) of each value is only equal to zero when the joystick is up (straight up or diagonally up)? And the second bit is only zero when it's down, the third when it's left, and the fourth when it's right. So we get Figure 15.

And that's why the numbers don't make sense when you first look at them.

---0	means "up"
---1	means "not up"
--0-	means "down"
--1-	means "not down"
-0--	means "left"
-1--	means "not left"
0---	means "right"
1---	means "not right"

FIGURE 15. Joystick bit chart

Here are a couple of machine language routines to help you make a little more sense out of the joystick values. One looks for vertical movement and will return a zero for up, one for center, and two for down. The other looks for horizontal movement and returns a zero for left, one for center, and two for right. As you'll see from the following example, these values can prove to be very practical:

```
100 DIM STICKV$(19), STICKH$(22)
110 FOR CHAR=1 TO 19
120 READ CODE
130 STICKV$(CHAR,CHAR)=CHR$(CODE)
140 NEXT CHAR
150 FOR CHAR=1 TO 22
160 READ CODE
170 STICKH$(CHAR,CHAR)=CHR$(CODE)
180 NEXT CHAR
200 GRAPHICS 0:POKE 752,1
210 PRINT :PRINT "Machine language joystick example"
220 POSITION 10,4:PRINT "VERTICAL VALUE:"
230 POSITION 8,6:PRINT "HORIZONTAL VALUE:"
240 VERT=USR(ADR(STICKV$),0)-1
250 HORZ=USR(ADR(STICKH$),0)-1
260 POSITION 26,4:PRINT VERT;"    "
270 POSITION 26,6:PRINT HORZ;"    "
280 GOTO 240
290 GOTO 250
```

```
1000 DATA 104,104,133,213,104,170,189,120,2,41,3
1010 DATA 201,2,240,1,74,133,212,96
2000 DATA 104,104,133,213,104,170,189,120,2,74,74
2010 DATA 73,2,201,3,208,2,169,2,133,212,96
```

If you wanted to read joystick one instead of joystick zero, you'd use USR(ADR(STICKV$),1) and USR(ADR(STICKH$),1).

Here's the assembly code that's stored in the DATA statements:

```
68    STICKV PLA
68           PLA
85D5        STA $D5
68           PLA
AA           TAX
BD7802       LDA STICK0,X
2903         AND #$03
C902         CMP #$02
F001         BEQ DONE
4A           LSR A
85D4  DONE   STA $D4
60           RTS

68    STICKH PLA
68           PLA
85D5         STA $D5
68           PLA
AA           TAX
BD7802       LDA STICK0,X
4A           LSR A
4A           LSR A
4902         EOR #$02
C903         CMP #$03
D002         BNE DONE
A902         LDA #$02
85D4 DONE    STA $D4
60           RTS
```

STICK0 is a shadow register for the last four bits (the leftmost four) of PORTA at location 54016. It is set to a value other than 15 when a light pen in the leftmost controller jack is pressed on the screen.

STICK1
633   0279

Same as STICK0 except it's a shadow register for the first four bits of PORTA rather than the last two. It's also, of course, the value for joystick one rather than

joystick zero.

STICK2
634 027A

Same as STICK1 except it's for joystick two, and it's also a shadow register for the last four bits of PORTB (54017).

STICK3
635 027B

Joystick three (the rightmost one) value and a shadow register for the first four bits of PORTB.

PTRIG0
636 027C

If you press the trigger on paddle zero, PTRIG0 will have a value of zero. If you don't press it, PTRIG0 will have a value of one.

PTRIG0 through PTRIG3 get their values from bits two, three, six, and seven of PORTA (54016), respectively. Because these are the same bits that tell whether joysticks one and two are moved to the right or left (see STICK0), you can use the trick in Figure 16.

PTRIG(1)-PTRIG(0)	= -1 if joystick zero is moved to the left
	=0 if joystick zero is in the center
	=+1 if joystick zero is moved to the right

FIGURE 16. PTRIG chart

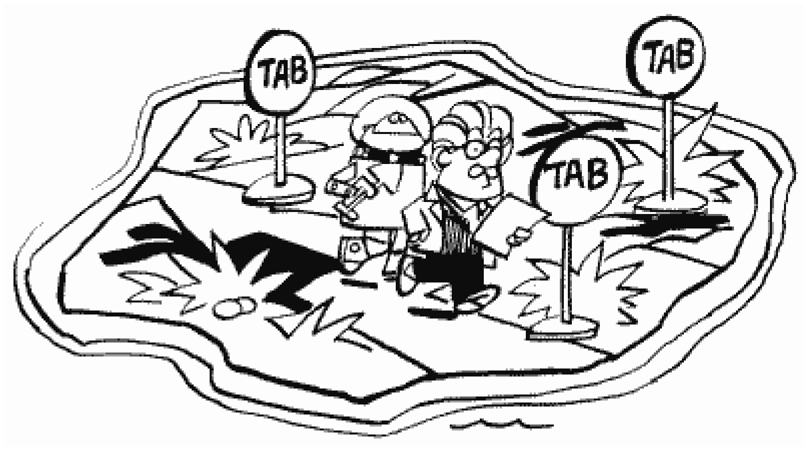
The same holds true for PTRIG(3)-PTRIG(2) and joystick one. You can use this trick to make horizontal movement easier to program. Just add the value of the PTRIG difference to your old horizontal position. This saves trying to figure out the joysticks. For the same ease in vertical movement, use the routine given for STICK0.

PTRIG1
637 027D

Same as PTRIG0 but for paddle one.

PTRIG2
638 027E

Trigger value for paddle two.



PTRIG3
639 027F

Trigger value for paddle three.

PTRIG4
640 0280

Trigger value for paddle four.

PTRIG4 through PTRIG7 get their values from bits two, three, six, and seven of PORTB (54017), respectively. The same trick for horizontal movement that was described under PTRIG0 can be applied to joystick two and joystick three using PTRIG4 through PTRIG7.

PTRIG5
641 0281

Trigger value for paddle five.

PTRIG6
642 0282

Trigger value for paddle six.

PTRIG7
643 0283

Guess.

STRIG0
644 0284

Well, here we are again at another new, different, and challenging name for a location. For the next three as well, actually. All three STRIG locations hold the values for the joystick button, and work exactly the same way as PTRIG (zero means pressed).

The STRIGs are shadow registers for the TRIGs (53264 to 53267).

STRIG1
645 0285

STRIG2
646 0286

STRIG3
647 0287

CSTAT
648 0288

CSTAT is the cassette status register.

WMODE
649 0289

This location tells the cassette handler whether the cassette is to be read from (0) or written to (128).

BLIM
650 028A

When the cassette handler reads in a record, the 132 bytes in that record are stored in CASBUF (1021). BLIM tells how many of those bytes are data that we want to give to the user. It is set according to one of the control bytes in the record, and since this probably doesn't make any sense to you, you should go read the description of CASBUF if you want more information.

Noname
651-655 028B-028F

More spare bytes that you shouldn't use because future versions of the OS might use them. Locations 651 and 652 are already used by version "B" as part of the interrupt handler routines.

The display handler uses the next 48 locations. Note that not all locations are used in all graphics modes.

1n the case of a graphics mode with a text window, the display handler takes care of the screen, while the screen editor takes care of the text window. Two separate IOCBs are used for this purpose (see locations 832 to 959), along with two separate cursors.

You should look at SWPFLG (123) for additional information about locations 656 to 667.

TXTROW
656 0290

The row that the text window cursor is currently in. Because there are only four rows in the text window, TXTROW ranges from zero to three.

TXTROW is the text window equivalent of ROWCRS at location 84.

TXTCOL
657,658   0291,0292

The column that the text window cursor is currently in. There are 40 columns in the text window, so TXTCOL can range from 0 to 39. "Ahah," you say. That means location 658 never gets used (since it's only needed when the column number is greater than 255). This is true under normal circumstances, but if you change the text window to be something other than graphics zero, you may need it.

TINDEX
659   0293

While we're on the subject of changing the text window graphics mode, TINDEX tells the OS what graphics mode the text window is (also see DINDEX at location 87). If you decide you'd like a different text window, you'll have to change the display list as TINDEX. Use the program for location SDLSTL (560,561) to look at the display list and see where the text window is. I won't go over it here because for most uses you'll probably just want to mix graphics modes. If that's not the case, however, it's easy to figure out how to make the necessary changes. Just look for the CHR 2 commands at the end of the display list. See location 559 and the appendices for more information.

TXTMSC
660,661   0294,0295

The address of the upper left-hand corner of the text-window screen memory. See SAVMSC (88,89) for the address of regular screen memory.

TXTOLD
662-667   0296-029B

Check out locations 90 through 95, OK? These six locations are the text-window equivalent, so I won't bother explaining them again.

TMPX1
668   029C

This, along with the next three locations, is used for temporary storage. They are used in one or more of the computer's routines as a place to store information during the routine. Once the routine is over, the values in them are no longer meaningful.

TMPX1, in case it wasn't clear, is a temporary location.

HOLD3
669   029D

A temporary location (the location isn't temporary, its use is).

SUBTMP
670   029E

Another temporary location.

HOLD2
671   029F

And yet another temporary location.

DMASK
672   02A0

Way, way back at location SHFAMT (111), we had a little discussion about masking and making changes to individual pixels in the graphics modes. Remember? Well, go back and refresh your memory anyway.

DMASK holds the mask for the pixel that we want to make changes to. Somewhere way up near the end of the OS ROM, there is a list of all the possible masks. The display handler decides which one is needed and loads it into DMASK. Here are all the different values DMASK can have, as well as the graphics modes they are used with (Figure 17).

By way of explanation, the "1's" are used to look at individual bits and the "0's" to ignore them.

Now why, you may ask, do we need more than one mask for most graphics modes? Graphics modes need anywhere from one to eight bits to represent a character or a pixel. Suppose a particular mode, such as mode nine, needs four bits per pixel. That means that each byte holds two different pixels, right (since a byte is eight bits)? So we need two masks to be able to mask out either pixel. This may be a little confusing to you, but don't worry. Unless you're programming in machine language, it's something that is nice to know but that you'll never need.

11111111	for modes zero, one, and two.
11110000	for modes nine, ten, and eleven.
00001111	
11000000	
00110000	for modes three, five, and seven.
00001100	
00000011	
10000000	
01000000	
00100000	
00010000	for modes four, six, and eight.
00001000	
00000100	
00000010	
00000001	

FIGURE 17. DMASK bit chart

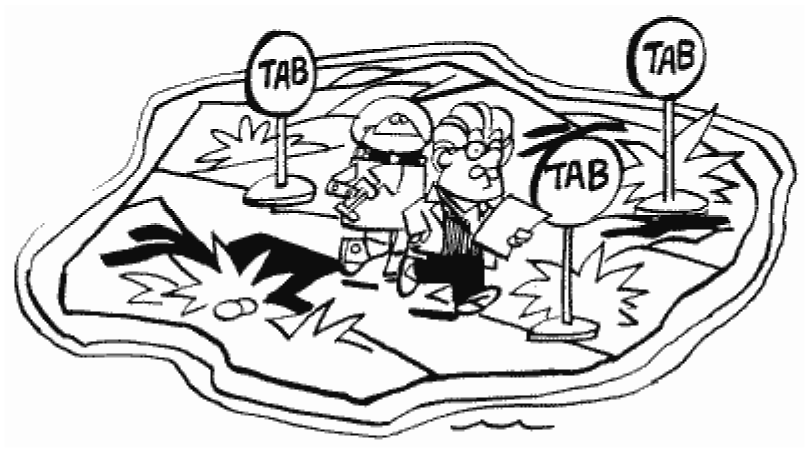
TMPLBT
673 02A1

More temporary storage space.

ESCFLG
674 02A2

When the ESC key is pressed, ESCFLG is set to 128 and the next key pressed gets an ESC flag attached to it (for example, pressing ESC twice would cause the second ESC to print a special character on the screen). After the next key has been pressed ESCFLG is reset to 0.

ESCFLG is initialized to zero.



TABMAP
675-689 02A3-02B1

TABMAP tells the OS what columns to move the cursor to when the TAB key is pressed.

When the TAB key is pressed, the cursor is moved to the next column, after the one the cursor is on, that has a tab stop. What's a tab stop? It's nothing more than a flag saying, "Hey, TAB, stop here, OK?" TABMAP is where these tab stops, or flags, are kept. Since you can set a tab stop on any one of the 120 columns in a logical line, TABMAP is 15 bytes long. What? How do you get 120 from 15? Well, since the tab stop for each column is either turned on or turned off, we only need one bit for each column. Fifteen bytes times eight bits per byte equals 120. Oh!

How do you set the tab stops? From BASIC, all you have to do is use the TAB-SET (SHIFT-TAB) and TAB-CLR (CTRL-TAB) keys. From within a program, however, you must change TABMAP yourself. To do this, start with 120 zeroes on a piece of paper. These represent the 120 columns, numbered from 0 on the far left to 119 on the far right. Now change the zeroes to ones in the columns where you want your tab stops. So far so good. The next step is to break the 120 digits into groups of eight and convert them to decimal. See the section on bits and bytes for help in doing this. The last step, now that you have 15 decimal numbers, is to POKE these numbers into TABMAP. You now have your own customized TAB settings.

What restores TABMAP to its original values? Pressing SYSTEM RESET or using a GRAPHICS command (OPENing S: or E: as well). What are its original values? A value of one in every byte. That translates to tab stops at 7, 15, 23, 31, … , 119.

A few final words. TABMAP works in graphics mode zero and in text windows only. Also, the left edge of the screen will always to be a tab stop, whether you set it to be or not.

LOGMAP
690-693 2B2-2B5

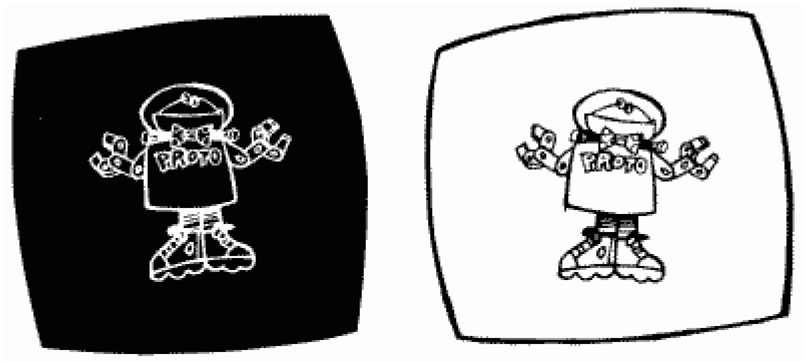
When you're writing or editing your BASIC program, the screen editor needs to know where each logical line begins. Why? Just so that it can make sense out of what's on the screen. Remember, a program listing on the screen may make sense to you, but to the computer it's just a bunch of bytes in memory. With the help of LOGMAP, at least it knows what row on the screen a program line begins on.

LOGMAP works in much the same way as the preceding TABMAP. There are 24 rows in a graphics zero screen, so there are 24 bits in LOGMAP. Actually, there are 32 bits (four bytes), but the last byte doesn't get used. The first byte handles rows 0 through 7, the second handles 8 through 15, and the third handles 16 through 23. If a logical line begins on a certain row, then the corresponding bit is set to one. If the row is part of a previous logical line, then the bit is set to zero.

All the bits in LOGMAP are set to one when the computer is first turned on, SYSTEM RESET is pressed, a GRAPHICS command is used, the text screen is OPENed, or the text screen is cleared. This is because all the lines are blank, and therefore considered to be the start of a logical line.

LOGMAP is updated when you first enter a logical line (with the RETURN key), edit a line, delete a line, or insert a line. Under all these circumstances, the position of the logical lines on the screen will be altered, thus the need for updating.

INVFLG
694 02B6



INVFLG works similarly to ESCFLG except it keeps track of the inverse video key (the Atari logo key) instead of the ESC key. It is initialized to 0, which means that all the characters you type in will be normal. But when you press the inverse video key, INVFLG gets set to 128 and characters that you type in now will appear in inverse video (black on white instead of white on black). Pressing the inverse video key again will restore INVFLG to 0 and get things back to normal.

Yon should be aware that changing INVFLG will only affect characters that are

typed in after you change it. That means that you can't use it like this:

POKE 694,128:PRINT "INVERSE?"

Machine language programmers might be interested to know that INVFLG is always XORed with the character value, regardless of INVFLG's value (this should tell you that the value for an inverse video character is just that for a regular character with bit seven set, i.e., the regular value plus 128). This means that you can have fun with the keyboard by POKEing INVFLG with something other than 0 or 128. Try it, it's fun !

FILFLG
695 02B7

If FILFLG is not equal to zero, then we're in the middle of a FILL.

TMPROW
696 02B8

A temporary storage place for the value in ROWCRS (84).

TMPCOL
697,698 02B9,02BA

More temporary storage, this one for the value in COLCRS (85,86).

SCRFLG
699 02BB

This one is somewhat complicated. First of all, it keeps track of how many physical lines (as compared to logical lines) have been scrolled off the top of the screen. If you keep pressing RETURN, it will eventually count up to 255 and then wrap back around to 0. No problem so far. According to the OS listing, however, it is also used during the character insertion process (when you press SHIFT-INSERT). Apparently, if you insert a character, SCRFLG gets set to 0. If the insertion caused the screen to scroll up, then the number of lines it scrolled (which depends on how long the logical line at the top of the screen was, so it could be from one to three) is stored in SCRFLG. The value in SCRFLG is then used to reposition the cursor, leaving SCRFLG with a final value of 255.

HOLD4
700 02BC

HOLD4 is used to temporarily hold the value of ATACHR (763) during the FILL routine.

HOLD5
701 02BD

Same as the above register (HOLD4).

SHFLOK
702 02BE

When SHFLOK is set to 0, all text typed in will be in lower case. Set it to 64, and all text will be in upper case. Finally, 128 will give you all control and graphics characters.

The following key combinations affect SHFLOK:

CAPS/LOWR sets it to 0.
SHIFT-CAPS/LOWR sets it to 64.
CTRL-CAPS/LOWR sets it to 128.

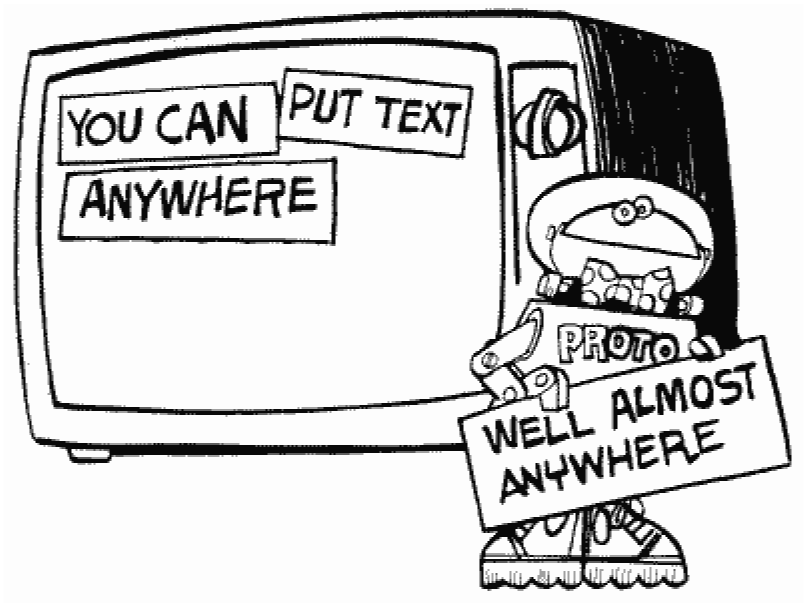
In addition, POKE in 192 and only numbers and punctuation will be recognized if pressed. Finally 255 in this location will not allow any letters at all to be recognized. Remember that these two POKES work on input from the keyboard only. You can still write letters to the screen or printer. This means their practical use is to prevent inputs you don't want.

Note that SHFLOK does not indicate whether or not the SHIFT or CTRL keys are pressed.

SHFLOK is initialized to 64.

BOTSCR
703 02BF

BOTSCR tells how many lines of text are available for use by the screen editor. What does this mean? Well, it can use all 24 lines in graphics mode zero, so BOTSCR would have a value of 24. In a mode with a text window, there are four lines in the text window that it can use, so BOTSCR would have a value of four. In all other modes there are none, so BOTSCR has a value of zero. What about graphics modes one and two? you say. In these modes the screen editor takes care of the text window, while the display handler takes care of the rest (at least in terms of PRINTing text, which is what we're really talking about here).



Try the following program:

```
100 GRAPHICS 0
110 POKE 703,4
120 FOR ROW=0 TO 19
130 POSITION 2,ROW
140 PRINT #6;"We have to print #6 up here"
150 NEXT ROW
160 PRINT "But now we have a text window here!"
170 PRINT
180 GOTO 160
```

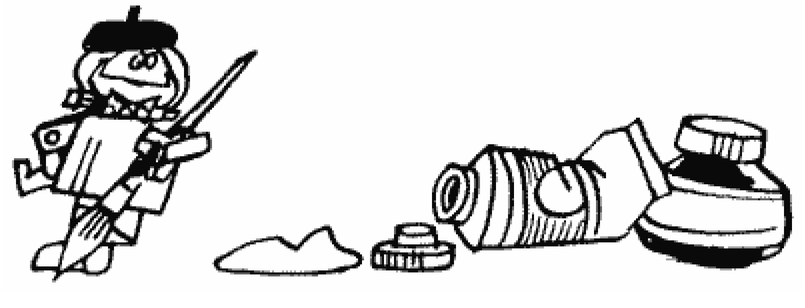
COLOR

The next nine locations, 704 through 712, are called “color registers.” This is just a fancy way of saying that they tell ANTIC what colors to put on the screen. How do you convert a color into a number that you can store here? The Atari has a total of 16 colors that you can choose from, and each is assigned a number. The exact colors vary slightly from television set to television set, so it’s difficult to describe them exactly. Bear that in mind when you consult the chart in Figure 18.

COLOR	VALUE	COLOR	VALUE
Black	0	Blue	8
Rust	1	Deep blue	9
Reddish orange	2	Dull blue	10
Dark orange	3	Olive green	11
Red	4	Green	12
Purplish blue	5	Dark green	13
Cobalt blue	6	Orangey green	14
Ultramarine	7	Orange	15

FIGURE 18. Color value chart

OK, now somewhere in the back of your mind you’re probably thinking “Wait a minute, aren’t there supposed to be 256 possible colors on the Atari?” Yes and no. There are only 16 colors, but there are also 16 shades of each color, resulting in a total of 256 (16 times 16) possible “colors.” That’s not even true either. Even though you can specify a brightness value from 0 to 15, 0 and 1 will be the same brightness, as will 2 and 3 and so forth. That gives us a true total of 128. 128 combinations of color and brightness, however, will be more than you need, or can use at one time.



So now we have a color value and a brightness value. Since each color register is only one byte, we’re obviously going to have to somehow combine these two values together. If you’re familiar with hexadecimal, you will probably know how already. Recall that in hexadecimal, each byte has two digits, each of which can have a value from 0 to F (15). All we do to combine our color and brightness is to have the first digit be the color, and the second the brightness. If you're using decimal, you want to multiply the color value by 16 and add the brightness. That's how you figure out the value to POKE into the appropriate color register (you can also use the SETCOLOR command for the playfield registers).

PCOLR0
704 02C0

This is the color register for player zero and missile zero. It is also used to hold the background color in GTIA mode 10.

The SETCOLOR command will not work on this or any of the next three locations.

PCOLR0 is a shadow register for COLPM0 at location 53266.

PCOLR1
705 02C1

The color register for player one and missile one. It is a shadow register for COLPM1 at location 53267.

PCOLR2
706 02C2

The color register for player two and missile two. It is a shadow register for COLPM2 at location 53268.

PCOLR3
707 02C3

The last player/missile color register, this time for player three and missile three. It is a shadow register for COLPM3 at location 53269.

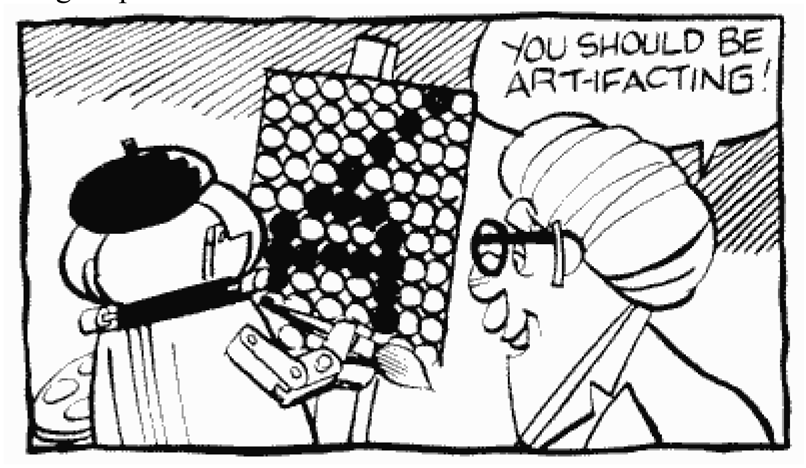
COLOR0
708   02C4

Lots of information for this guy. This is the color of playfield zero. It is also called color register zero, is a shadow register for COLPF0 at location 53270, specifies the color of uppercase letters in graphics modes one and two, and can be set by the BASIC SETCOLOR command (as can the next four locations). Whew!

COLOR1
709   02C5

This holds the color value for playfield one, is called color register one, is a shadow register for COLPF1 at location 53271, and specifies the color of lowercase letters in graphics modes one and two.

COLOR 1 is also used to specify the brightness of the characters in graphics mode zero, and of the pixels in graphics mode eight. As you know, you can only draw with one color in graphics mode eight, right? Well, not quite. Through a process called "artifacting," you can get up to four.



Briefly, because the pixels are so small in graphics mode eight, a pixel in an odd-numbered column will have a different color than one in an even-numbered column. Don't ask why, just try the following program:

```
100 GRAPHICS 8
110 COLOR 1
120 FOR COL=10 TO 20 STEP 2
130 PLOT COL,10
140 DRAWTO COL,20
150 NEXT COL
160 FOR COL=31 TO 41 STEP 2
170 PLOT COL,10
180 DRAWTO COL,20
190 NEXT COL
```

Viola! Two new colors. But how do we get the regular white, and where does the fourth color come from? You know, you ask a lot of questions. If we plot an even-numbered column and then the following odd-numbered column, we get white. If, on the other hand, we plot an odd-numbered column and then the following even-numbered one, we get the fourth color. Make sure you understand the difference between the two. Add the following lines to the preceding program:

```
200 FOR COL=50 TO 60 STEP 4
210 PLOT COL,10:DRAWTO COL,20
220 PLOT COL+1,10:DRAWTO COL+1,20
230 NEXT COL
240 FOR COL=71 TO 91 STEP 4
250 PLOT COL,10:DRAWTO COL,20
260 PLOT COL+1,10:DRAWTO COL+1,20
270 NEXT COL
```

Doing things this way kind of restricts you in the way you plot and draw, but it does give you more colors. You should also note that the CTIA and GTIA chips switch the odd and even colors on the screen. This usually makes red on one computer look like green on another. Also, the colors you do get will depend on the values in COLOR1 and COLOR2 (following).

COLOR2
710 02C6

This holds the color value for playfield two, is called color register two, is a shadow register for COLPF2 at location 53272, and specifies the color of inverse uppercase letters in graphics modes one and two.

In graphics modes zero and eight, COLOR2 specifies the background color.

COLOR3
711 02C7

OK, you should be getting the hang of this by now. This is the same as COLOR2, but with threes instead of twos. It is a shadow register for COLPF3 at location 53273. It’s also the color of inverse lowercase letters in graphics modes one and two.

COLOR4
712 02C8

Same as the preceding but for the background color. It is a shadow register for COLBK at location 53274. Don’t forget that in GTIA modes, PCOLR0 (704) is the background color, while COLOR4 is just a regular color register.

Noname
713-735 02C9-02DF

These 23 are currently unused.

The following four bytes, from 736 to 739, are used by DOS. That means that they are unused if you are not using DOS.

GLBABS
736-739 02E0-02E3

Global variables, or, four spare bytes for non-DOS users. For DOS users they are as used below.

RUNAD
736,737 02E0,02E1

When you load a binary load file from DOS, sometimes it will run automatically and sometimes it won't. What makes the difference? If the binary load file stores an address in RUNAD, then DOS will go (JSR) to that address after the file has been loaded. Otherwise, the DOS menu will stay on the screen. See your DOS manual for more information under the sections on binary loading and saving.

INITAD
738,739 02E2,02E3

Whoops! I lied slightly in the last location. If a binary load file alters INITAD, then DOS immediately goes (JSR again) to the address in INITAD before continuing to load the file. You can use this to do stuff like put a message or picture on the screen while the rest of the file is loading. Make sure that the routine whose address you're putting in INITAD ends with an RTS. Also, if you want DOS to return to the menu after executing the RUNAD routine, make sure it ends with an RTS instruction.

So where was the lie? If you don't end the INITAD routine with an RTS instruction, the RUNAD routine will never be executed (and you may run into problems with future disk I/O).

RAMSIZ
740 02E4

RAMSIZ has a similar function to RAMTOP (106), so go back and read up on RAMTOP. The main difference is that RAMSIZ doesn't cause the screen memory to move when you change it and do a graphics call. Experiment to see how it works.

MEMTOP
741,742 02E5,02E6

MEMTOP holds the address of the last free memory location. This does not mean the top of RAM. Why not? You're forgetting that the screen memory and display list are put at the end of RAM. MEMTOP is the last location that is unused, and is therefore the location right below the display list. Originally, however, before any graphics mode is set up, it does hold the same address as RAMSIZ.

Anything that results in the display handler changing the screen memory and display list also results in MEMTOP getting changed. That means SYSTEM RESET, the GRAPHICS command, and OPENing the screen.

For more information on MEMTOP's use (yes, I'm going to send you somewhere else again), see APPMHI at locations 14 and 15.

MEMTOP is called HIMEM by BASIC, since BASIC has its own MEMTOP at locations 144 and 145.

MEMLO
743,744 02E7,02E8

Since we have a pointer to the top of free memory, it only makes sense to have one to the bottom of free memory. MEMLO holds the address of the first byte in RAM that is available for your use. Notice that BASIC uses a different pointer for the first free byte, called LOMEM (128,129). Although some sources imply otherwise, MEMLO and LOMEM seem to always contain the same value, which is not touched by the OS after the power up routine is done.

The first free location in memory is usually at 1792. If you're using DOS, however, DOS needs some extra space for something called the "FMS buffers" (see SABYTE [1801], DRVBYT [1802], and the glossary). This means that MEMLO will be greater when DOS is present (by 128 for each buffer).

Let's talk about reserving memory for your own private use. We last discussed this at RAMTOP (106), where we saw how to reserve memory above screen memory. But, alas, this technique wasted up to 800 bytes because of a problem with scrolling the screen. So now we come to the alternative of reserving memory at the other end of RAM, below everything else. How do we do it? There are two possibilities. First of all, you could write an AUTORUN.SYS file that loads MEMLO with the values you want. De Re Atari has an excellent example of how to do this, but it's obviously a technique that requires a knowledge of machine language. What if you're working in BASIC? Well, there's a problem. Remember that BASIC also keeps a pointer to the bottom of free memory. It's called LOMEM and I have mentioned it. If we change MEMLO, we also have to change LOMEM. We can do this by POKEing both MEMLO and LOMEM, but that confuses BASIC because it loses some

important information that it had already stored in the memory area you just told it not to look at. That's a problem. What happens if you POKE MEMLO and then type NEW (NEW transfers the value of MEMLO into LOMEM and resets all the program pointers)? Nothing bad; in fact it does exactly what we wanted. But we still have a problem: this method only works when you make the changes yourself; it won't work from inside a program. As it turns out, and it makes sense if you think about it, there is nothing you can do from within a BASIC program to reserve memory using MEMLO (without destroying the program). This means that the MEMLO method of reserving memory is only useful if you're programming in machine language (or if you first boot up an AUTORUN.SYS file as described). Sorry folks.

SYSTEM RESET will restore MEMLO to its original value. The program in De Re Atari, as mentioned, uses the SYSTEM RESET vector to make sure that MEMLO does not get reset.

Only NEW (or turning off the computer) will restore LOMEM.

Noname
745 02E9

Currently unused. This is, however, subject to change in future versions of the OS.

DVSTAT
746-749 02EA-02ED

This one is for experts only, so don't expect it to sound pretty. When you send a GET STATUS command (83) to a device, these bytes are set according to the type of device and its status. They seem to be set only by the printer handler, the disk handler (not the disk file manager), and the RS-232 handler.

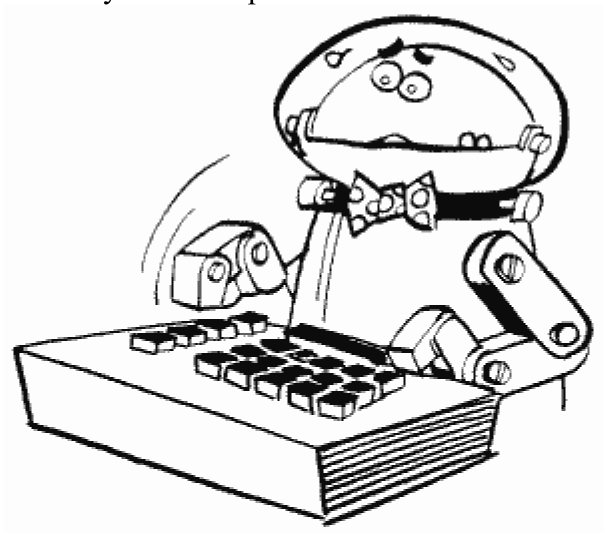
Location 746 gives the command status. Because it is interpreted differently for each device, you should consult either the OS manual or the 850 Interface manual for details (this isn't a cop-out on my part; the information in this byte is useful only to extremely competent machine language programmers).

If the GET STATUS were to a printer, location 747 contains the AUX2 byte of the previous operation. If it were to a disk drive, location 747 holds the value of the status byte of the disk controller chip (if you really need to know more details, find some documentation on the INS1771-1 Floppy Disk Controller chip). Finally, if it were to the 850 Interface, location 747 could indicate one of two things. If concurrent mode I/O is not active, then it will hold information regarding the monitored readiness lines (DSR, CTS, and CRX) and the data receive line (RCV) of the specified port. Please see your 850 Interface manual for more details.

If concurrent mode I/O were active, location 747, in conjunction with location 748, will hold the number of characters currently in the input buffer.

For the printer and disk drive, a GET STATUS command will return the maximum timeout value for the device. This value is provided by the device controller and is initialized to 31. A value of 60 here represents one minute.

Location 749 is only used for the 850 Interface, and only if concurrent mode I/O were active at the time of the GET STATUS. In that case, it holds the number of characters currently in the output buffer.



If you got this far and you're confused, don't worry. By the time you have a need to use DVSTAT, it should be easier to understand. I've been programming the Atari for five years and have only recently found a need for it.

CBAUDL, CBAUDH
750,751 02EE,02EF

The speed at which programs load in from cassette is called the "baud rate," and this is what is stored in CBAUDL/H. It's initialized to 1484 by the OS, which represents 600 baud ("baud," by the way, stands for "bits per second"; don't ask how they got one from the other). Unfortunately, sometimes the data on the cassette tape is stored a little slower or faster than 600 baud. This may be due to the speed of the cassette motor, the tape being stretched slightly, or other such minor details. In any case, at the beginning of each cassette record (remember that a record is just a bunch of bytes) are two bytes that have alternating zeroes and ones (01010101; 85). These bytes are used to set the baud rate exactly, so speed variations can be compensated for.

AUDF3 (53764) and AUDF4 (53766) are used to store the baud rate and do the actual timing.

CRSINH
752 02F0

This one should come as a reward to you for trudging through the sludge of the last few locations. CRSINH is used to make the cursor invisible (and visible again). This comes in handy when you've got a message or something on the screen and you don't want whoever's reading it to see the cursor. All you have to do is POKE CRSINH with something other than a zero. To make the cursor visible again, just POKE it with a zero. That's (almost) all there is to it.

Hold it, what was the "almost" that was trying to hide in the parentheses back there? Well, there is one tiny thing I forgot to mention. The cursor won't disappear (or reappear) until you move it for the first time after you change CRSINH. All that means is you have to have a PRINT of some kind after the POKE. The easiest way around this is just to POKE CRSINH before you print anything on the screen. For example,

```
100 POKE 752,1
110 PRINT
```

CRSINH is set to zero when you turn on the computer, and also when you press BREAK, press SYSTEM RESET, use a GRAPHICS command, or OPEN either "S:" or "E:".

Also see CHACT at location 755 for another way to tell the cursor to get lost.

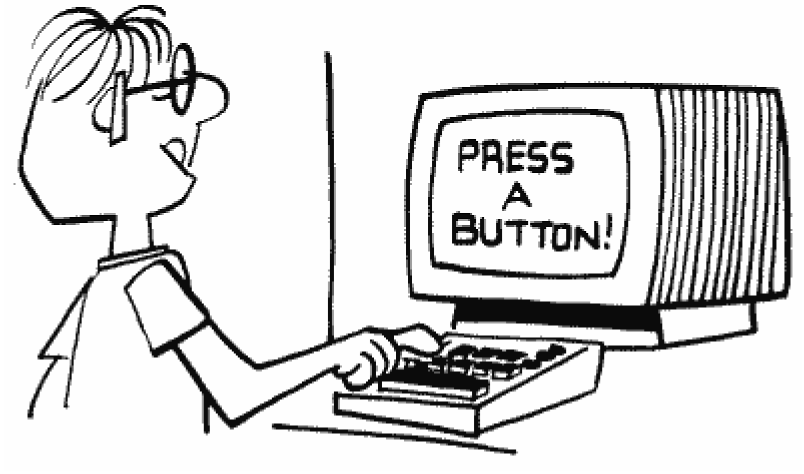
Here is a way to place dots all over your screen so that you can check the convergence of the TV or monitor:

```
10 POKE 710,0:POKE 752,1:POKE 82,0:FOR I=1 TO 959:? ".";:NEXT I
20 GOTO 20
```

KEYDEL
753 02F1

A lot of you have probably heard the term “debounce” (no, it’s not from a commercial for French shampoo). Some of you probably don’t have the slightest idea what it means, so let’s talk debounce for a bit.

When you press a key, you’re actually bringing two little bits of metal together. When the two touch, electricity flows through them and tells the computer that the key is pressed. Sometimes, when the two first hit each other, they bounce a little. This has the effect that they are touching, then they’re not touching, and then they’re touching again, which the computer would normally interpret as meaning the key was pressed twice. You only pressed it once, however, so somehow the computer has to be smart enough to realize this. The process it uses is called “debouncing” and it’s fairly simple. If a bounce occurs, it happens real fast, too fast for you to have been able to hit the key twice. So, the OS waits a little while after you first press the key before looking to see if you pressed it again. That way, it doesn’t see the bounce. KEYDEL tells it how long to wait.



KEYDEL is set to three whenever a key is pressed and then every stage two VBLANK it’s decremented by one. Until it reaches zero, the OS will not let the same key be pressed again. Unless you can press a key faster than 20 times a second, this won’t be a problem for you.

CH1
754   02F2

CHI is the value of the last key pressed (not the current one). When you press a key, the OS checks its value (stored in CH [764]) against CH1. If they're the same, then KEYDEL is checked to make sure that the key has been debounced. If KEYDEL is equal to zero, or if the two values aren't the same, then the current key code is stored in CHI and the OS goes on to process that key.

CHACT
755   02F3

CHACT does some neat things to the characters on the screen. The bits are used as summarized in Figure 19.

-------0	inverse character letters are visible, but not background
-------1	inverse character letters are totally invisible
------0-	inverse character backgrounds are invisible
------1-	inverse character backgrounds are visible (default)
-----0--	all characters are right-side up
-----1--	all characters are upside down

FIGURE 19. CHACT bit chart

What does this mean? Try typing some inverse characters on the screen (use the Atari logo key). Now POKE 755,1. What happened? That's right, the letters disappear. Try POKE 755,2. This makes the background (the solid white part) disappear. Finally, try POKE 755,3 to make everything disappear (everything in the inverse characters, that is). That should give you a good idea of what the first two bits can do. By the way, since the cursor is essentially an inverse character, it will disappear as well when you make the inverse character background disappear.

The last bit is pretty self-explanatory. Just try POKE 755,4 and see what happens.

What can you use CHACT for? Reverse characters add emphasis to text, CHACT lets you add even more emphasis by making inverse characters blink. Try the following:

```
100 GRAPHICS 0
105 POKE 752,1
110 PRINT:PRINT "Add EMPHASIS to your programs"
120 FOR BLINK=1 TO 10
130 POKE 755,0
140 FOR DELAY=1 TO 50
150 NEXT DELAY
```

```
160 POKE 755,2
170 FOR DELAY=1 TO 50
180 NEXT DELAY
190 NEXT BLINK
195 POKE 752,0
```

Try substituting other values for the zero in line 130. Also see location VVBLKD at 548 and 549 for a machine language routine that uses CHACT to make inverse text blink while you're typing it.

In case you hadn't figured it out already, CHACT is initialized to two.

CHBAS
756 02F4

This is a biggie (have I ever lied?). CHBAS holds the address, in pages (so you multiply the number here by 256 to get the actual address), of the character set. What is a character set? A character set is a whole bunch of numbers that tell the computer how to draw the various characters on the screen. In other words, it tells the computer what the characters look like. How can numbers describe what a character looks like? First of all, you should go read the section near the beginning of the book on bits and bytes. Then come back here.

Back already? OK, what do bits and bytes have to do with character descriptions (why am I asking so many questions)? Well, a byte can be thought of as part of a picture. You know – with the bits being dots in the picture. You turn a bit on, and the corresponding dot in the picture gets turned on. You've already seen how this is used in the graphics modes. Well, the text modes also need to turn dots on and off, but they need to change a whole bunch at once for each character. So what the Atari does is store eight bytes for each character in this special thing called the character set. Each of these descriptions is given a number, and to set the right dots for a particular character, the computer just has to say, "Hey, get me the description for character number whatever and put it on the screen," and the character will magically appear on the screen.

Let's take a look at how those eight bytes make up a character (Figure 20).

# IN	BIT
MEMORY	PATTERN
0	00000000
12	00001100
28	00011100
60	00111100
108	01101100
126	01111110
12	00001100
0	00000000

FIGURE 20. CHBAS bit chart

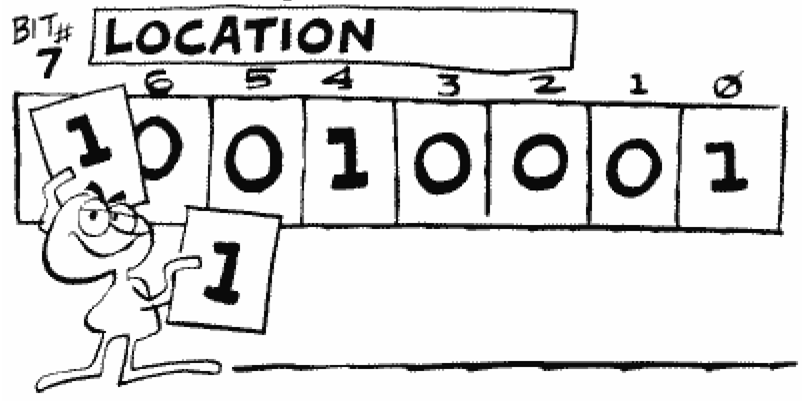
Now you can see how simple creating characters is. First draw the shape of an 8 x 8 pattern of 0's and 1's. Next add up the value of the ON, or 1, bits. Then POKE these numbers in the proper order into memory. Let's go over the details.

Look at those bits again in terms of dots, with the zeroes meaning no dot, and the ones meaning dot (Figure 21).

```
    ##
   ###
  ####
 ## ##
 ######
    ##
```

FIGURE 21. The number 4

Ahah! The description we used was for the "4" character. You should now be able to see how the descriptions work.



How are the descriptions ordered within the character set? It's not the same order as ATASCII (the order that CHR$ and ASC use). To convert from ATASCII values, which you can find in your BASIC manual, to the character set order, use Table 2.

TABLE 2

TYPE OF CHARACTER	ATASCII NO.	CHAR. SET NO.
uppercase, numbers, punctuation	32-95	0-63
graphics, characters	0-31	64-95
lowercase	96-127	96-127

Now, to find the character description of a particular character, find the ATASCII value (either with ASC or by looking it up in the chart in the BASIC manual), use the preceding chart to convert it to the character set value (more commonly called the "internal" value), multiply that by eight (because there are eight bytes for each character), and add it to PEEK(CHBAS)*256. The result is the address of the first byte of the character description you want.

The character set that comes with the Atari is stored starting at location 57344. You can double-check this by PEEKing CHBAS and seeing that it has a value of 224. There are a total of 128 possible characters (not counting inverse ones), so the character set takes up 128*8 equals 1024 bytes.

In graphics modes one and two, you probably know that you can't have upper- and lowercase letters on the screen at the same time. Why not? In these modes the characters can be one of four possible colors. In order to be able to pull this off, two of the bits in the character number have to be used to specify the color. This means that only six bits are left to specify the character. Six bits are enough to give you the numbers 0 through 63. Zero through 63, if you consult the preceding character order chart, are the uppercase characters, numbers, and punctuation. So what if you want lowercase? The BASIC manual tells you to POKE 755 (CHBAS) with 226 instead of 224. What does this do? It moves the start of the character set forward by 512 (2*256) bytes. Now I know that right now you're thinking to yourself, "Gee, 0 through 63 is a total of 64 characters, and 8 bytes for each character gives me, let me see, uh, 512 bytes!" Hey, you're terrific! What you just caught on to is that changing CHBAS like that simply lets you skip over to the lowercase and graphics characters, the other half of the character set.

Unfortunately, this means that the heart character gets used as a space, so your screen is filled with hearts-romantic, but not what you want. You can get rid of them with SETCOLOR 0,0,0 or by redefining the heart character to a space. See Appendix One for information on how to do the latter.

In graphics mode zero, there's not much more to tell. If an inverse video character is requested (see INVFLG [694]), then the eight bytes for that character are reversed (ones changed to zeroes and vice versa) before they are put on the screen.

CHBAS is a shadow register for CHBASE at location 54281. For some reason **you cannot set CHBAS to an odd number**, or garbage will fill the screen. Finally, CHBAS can be set to point to your own character set.

"Hold on there, just a second, wait a minute, timeout, take five, whoa! You mean I can design my own character set? And you took all this time before you told me, and now you're going to move on without telling me how to do it? What kind of author are you?"

By the way, please see Appendix One for a complete example of designing your own character set.

Noname
757-761 02F5-02F9

More spare bytes. You know, I have to assume that you're going to come to these locations and forget all about that warning I gave you way back when. You remember, "Don't use spare bytes, they may be used in future versions of the OS." But if you did remember and are getting sick of me telling you every time we come across some spare bytes, then what can I say? It's a rough world out there.

CHAR
762 02FA

This is the internal number (value) of the character that was read or written last by the display handler. A lot of the time the handler will move the cursor as the last step of an operation, so PEEKing here will often return a value of 128 or 0 (for a visible or invisible cursor respectively).

ATACHR gives the ATASCII value corresponding to the internal value in CHAR.

ATACHR
763   02FB

ATACHR is used by the display handler, the screen editor, and the keyboard handler to hold the ATASCII value of the character last read or written. If we're using a graphics mode rather than a text mode, then it's the value of the graphics byte rather than that of the character (for the display handler only). It's also used in converting ATASCII to internal and vice versa, and FILL uses it to hold the color of the area being filled (in which case it gets its value from FILDAT [765]).

CH
764   02FC

CH is the middle guy between the keyboard and the keyboard handler. When a key is pressed, a keyboard value (yes, yet another kind of character value) gets put into CH. The keyboard handler then picks it up, puts it into CH1 (754), and puts a 255 into CH to indicate that it got the value OK. There are a few exceptions to this. First of all, if we're in the middle of debouncing (see KEYDEL [753]) the key is ignored completely; it doesn't even make it to CH. If CTRL-1 is pressed, then SSFLAG (767) is updated, but CH is not affected. Finally, CH also gets changed by the key repeat process mentioned under SRTIMR at location 555. To repeat a key, the OS takes the value in KBCODE (53769) and stores it in CH.

If you are GETting information from the keyboard, make sure you set CH to 255 before you do your GET. This will make sure that any previous key presses are ignored. For example,

```
100 OPEN #1,4,0,"K:"
110 POKE 764,255
120 GET #1,A
130 PRINT "You pressed key number ";PEEK(754)
140 GOTO 110
```

You can use this program to find out the values for the various keys, or you can look at the chart on page 50 of the OS manual. In either case, you should notice that the CTRL key adds 128 to a key value, and the SHIFT key adds 64.

Here is my favorite trick for this location. Say you want your program to load in a tape program and then RUN it. It would seem that there is no way to do that because someone has to press the RETURN key after the program loads and you type RUN. NOT TRUE. Use location 764 to hold the RETURN key like this:

10 REM YOUR PROGRAM HERE
.
.
.
.
2000 POKE 764,12:CLOAD:RUN

FILDAT
765 02FD

Simply put, FILDAT is the data to FILL with in the XIO 18 command.

DSPFLG
766 02FE

When DSPFLG is set to a nonzero value, then CTRL characters like CTRL-CLEAR, CTRL-DELETE, CTRL-arrow, and so forth will appear as a character on the screen rather than having some kind of effect on the screen (such as clearing it or moving the cursor). If it's equal to zero, then they have their normal effect.

Note that to type a CTRL character so that it appears on the screen, you press ESC before you type that character. ESCFLG (674) is ORed with DSPFLG before the character is processed. That means that the ESC key is not the only way to get CTRL characters to appear. That's good. Suppose, for example, that you want to print the arrow characters on the screen from BASIC. You can use the ESC key to type them into a string, but when you try to print that string to the screen, BASIC will move the cursor rather than print the arrows. What you have to do is POKE 766,1 before you try and print the string. Be sure to change it back afterwards.

SSFLAG
767 02FF

SSFLAG is used to pause a program or a LISTing. When it's set to 0, everything works as usual. When set to 255, however, the pause is in effect and will stay that way until it's set back to 0 again. If the basic idea of this sounds like something you've run across before, that's because it is. The CTRL-1 key, which you have probably used to pause your LISTings, changes SSFLAG. You can also change SSFLAG yourself, but if you do it from within a BASIC program, keep in mind that the program is paused, so you won't be able to change it back unless somebody presses CTRL-1! Try this:

```
100 POKE 202,1
110 PRINT "Now try LISTing this program"
```

SSFLAG has no effect on machine language routines, which is why you can't use CTRL-1 to pause some programs.

PAGE THREE

You probably aren't going to be too thrilled with page three. Why? It's all about I/O. That means that you may not understand a lot of it, because I/O can get real complicated real fast. Don't worry too much about it, though. BASIC has commands that take care of these locations for you, so you're only reading about these locations for enlightenment. If, on the other hand, you're programming in machine language…

Before you go any further, make sure you've read and at least vaguely understood the appendix on I/O. It's not that long, or complicated, but it will give you a nice overview of what everything here is used for.

The first part of page three, locations 768 through 831, is used for the "device handlers." As the name implies, device handlers are used to handle I/O to the various devices. What devices can we have? The screen (S:), the screen editor (E:), the keyboard (K:), the cassette player (C:), the disk drive (D:), the printer (P:), and the RS-232 ports on the 850 interface (R:); all of these handlers, which are just machine language routines, are a part of the OS, with the exception of the RS-232 handler. The RS-232 handler is stored inside the 850 interface, and gets transferred over to the Atari when you turn on the system.

Locations 768 through 779 make up the Device Control Block (DCB; see the appendix on I/O for an explanation). To use the DCB you must set it with the appropriate values and then JSR DSKINV (58451) for disk I/O, or JSR SIOV (58457) for other device I/O.

DDEVIC
768 0300

Three of the devices, S:, E:, and K:, are a part of the computer. The others are all outside the computer, and we therefore need to have some way of talking to them. The "serial bus" takes care of that (the cords you use to connect the devices together are the visible part of the serial bus). But, since you can have more than one device hooked up to the serial bus, you need some way of telling the bus which device you want to talk to. Each device is therefore assigned a number (think of it as a phone number), and the handler gives DDEVIC the number of the device it wants to talk to. Do not change DDEVIC yourself.

Here are the numbers for the various devices (Figure 22).

Disk Drive	D1-D4	49-52 ($31-$34)
Printer 1	P1	64 ($40)
Printer 2	P2	79 ($4F)
RS-232 Ports	R1-R4	80-83 ($50-$53)
Cassette	C	96 ($60)

FIGURE 22. DDEVIC chart

DUNIT
769 0301

We can have up to four disk drives and RS-232 ports. DUNIT holds the number of the disk drive, printer, or RS-232 port we want.

DUNIT gets added to DDEVIC, and the result stored in CDEVIC at location 570. CDEVIC is then used during the actual I/O.

DCOMND
770 0302

Once it has got the number of the device we want to talk to, the handler has to know what it should tell the device to do. For that we have another bunch of numbers, this time for the various commands (Figure 23).

Get Sector	82 ($52)
Put Sector (with verify)	87 ($57)
Put Sector (w/o verify)	80 ($50)
Get Status	83 ($53)
Format Disk	33 ($21)
Download	32 ($20)
Read Address	84 ($54)
Read Spin	81 ($51)
Motor On	85 ($55)
Verify Sector	86 ($56)

FIGURE 23. DCOMND chart

This is one of those tables that gives you the confidence that you know what's going on, until you get about halfway down. The first five commands are probably the only ones you'll ever run into, so don't worry too much about it.

DDEVIC gets transferred over to CDEVIC (570) for use by SIO.

DSTATS
771 0303

Two uses for DSTATS. First of all, after an I/O operation is complete, it holds the status of the operation. A zero means that everything went OK. See the OS manual for the meaning of nonzero values.

Before the I/O operation, DSTATS tells SIO how data is going to be transferred, using bits six and seven as in Figure 24.

00------ ($00) means no data will be transferred in this operation.
01------ ($40) means data is going to be read from the device.
10------ ($80) means data is going to be written to the device.
11------ ($00) is not a valid combination.

FIGURE 24. DSTATS chart

DBUFLO, DBUFHI
772,773 0304,0305

This is a pointer to the buffer that will be used to store the data that is to be sent or received during I/O. It's set by the handler to the system buffer at CASBUF (1021) unless you tell the handler differently.

If a GET STATUS command is given, then DBUFLO/HI is set to point to DVSTAT (746).

If you're communicating with SIO directly, you should make sure you set DBUFLO/HI yourself.

DTIMLO
774 0306

DTIMLO is the timeout value for the device being used and is set by the handler. You will recall from our other run-ins with timeouts that a value of 60 here represents 64 seconds.

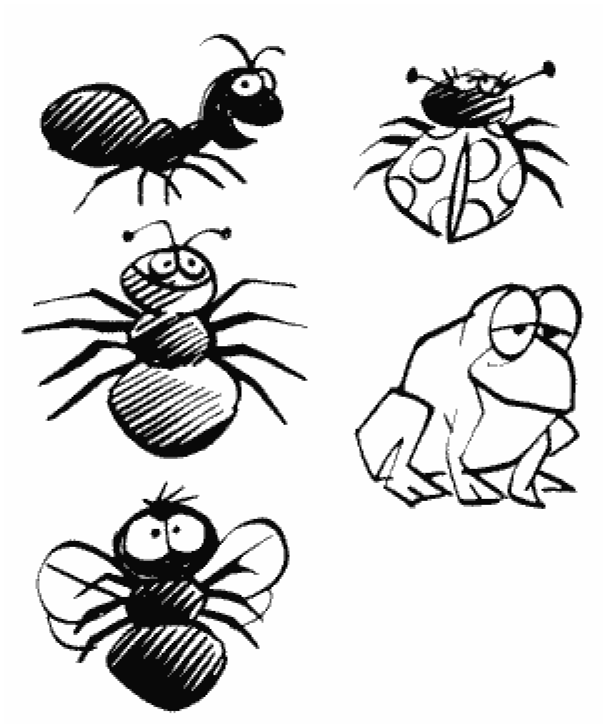
DTIMLO is initialized to 31.

DUNUSE
775 0307

Another unused byte (warning, warning!).

DBYTLO, DBYTHI
776,777 0308,0309

This location specifies the number of data bytes that are to be read to, or written from, the buffer during I/O. It is also used by the FORMAT command to store the number of bad sectors.



The values in DBUFLO/HI and DBYTLO/HI are added together after the I/O is over, and the results stored in BFENLO/HI (52,53).

Just in case you thought the OS was perfect, there's a bug that messes things up if the last byte in the buffer is in an address that ends in $FF (such as $41FF, $32FF, etc). Be careful about this.

DAUX1, DAUX2
778,779 030A,030B

These are used to provide information that is unique to the specific device (a sector number, for example). Their values are transferred to CAUX1 and CAUX2 at locations 572 and 573.

The next 14 locations (780 to 793) have various SIO uses.

TIMER1
780,781 030C,030D

TIMER1 is the initial timer value for the baud rate. What does that mean? Back at CBAUDL/H (750,751) we discovered what a baud rate is and how the OS constantly adjusts it during I/O. We found out that an alternating bit pattern is read and timed in order to figure out the correct rate. TIMER1 stores the time at the beginning of this pattern, and TIMER2 below stores the time at the end of it. The difference in these times is then used to figure out the new baud rate.

The first byte of both TIMER1 and TIMER2 is the value of VCOUNT (54283) at the time, and the second is the value of RTCLOK+2 (20).

ADDCOR
782 030E

ADDCOR is an “addition correction flag” used in the baud rate calculations. Those quotation marks mean that you’ll never need to know what it means.

CASFLG
783 030F

Part of the SIO routine is not needed for cassette I/O, so CASFLG is used to warn SIO that cassette I/O is being done. A value of 0 means regular SIO, 255 means cassette.

TIMER2
784,785 0310,0311

This is the final timer value for baud rate. See TIMER1 for a complete description.

TEMP1
786,787 0312,0313

TEMP1 is used as a temporary storage location for the difference in the TIMER1/2 values during the baud rate calculation.

TEMP2
788 0314

Supposedly another temporary storage location of some sort, but according to the OS listing it isn’t used.

TEMP3
789 0315

Another temporary storage location that is used, but for nothing particularly important.

SAVIO
790 0316

Back to setting the baud rate. Remember the alternating bit pattern (see TIMER1 if not)? SAVIO is used to check the serial port SKSTAT (53775) to see if the next bit has come in yet. That's all.

TIMFLG
791 0317

This is a flag to indicate that the cassette player has timed out (taken a snooze). If it's equal to one, we're OK. If it's equal to zero, then we're in timeout territory.

For the cassette player to timeout, a data byte must not be found within the given time period (which can vary). This usually indicates that the baud rate was wrong, assuming that you remembered to connect the cassette player, plug it in, put in the program tape, and press "PLAY"!

STACKP
792   0318

Remember the stack at page one? When SIO first gets going, it stores the value of the stack pointer in STACKP. That way, if somebody presses BREAK before it's done, it can restore the stack pointer and return to where it was called from.

TSTAT
793   0319

This is used to temporarily hold the value of STATUS (48) during I/O.

HATABS
794-831   031A-033F

We now know a little about what handlers do, but where do we find them? Obviously HATABS is going to have something to do with it, but before I tell you what, let's talk a little more about handlers.

Each handler is made up of a bunch of routines that perform different I/O functions. These functions are shown in Figure 25.

OPEN device
CLOSE device
GET BYTE from device
PUT BYTE to device
GET STATUS of device
SPECIAL
INITIALIZE device

FIGURE 25. I/O functions chart

Since SIO is going to need to know where each of these routines is, it's useful to keep the address of each routine in a table. We'll only need the initialization routine once, so we'll put a JMP instruction in the table right before the initialization address. Finally, we'll call this table the "handler entry point" table, which makes sense if you think about it.

OK, so now we have a handler entry point table for each of our handlers. Now we need a table of the addresses of these tables (aren't computers fun?). This is where HATABS comes in. Each entry in HATABS consists of the ATASCII value of the one character device name ("C", "D", "K", etc.), followed by the address for the handler entry point table for that device. So, keep in mind that even though HATABS is called the "handler address table," it is actually the handler entry point table address table!

When you first turn on the computer, five entries are automatically set up in HATABS. They are for the printer, cassette player, screen editor, screen, and keyboard handlers, in that order. If a disk drive is hooked up and turned on, then the entry for the disk handler is added. Finally, if the 850 Interface is hooked up and on, the entry for the RS-232 handler is added after that for the disk. The addresses for the handler address table of each of these are shown in Figure 26.

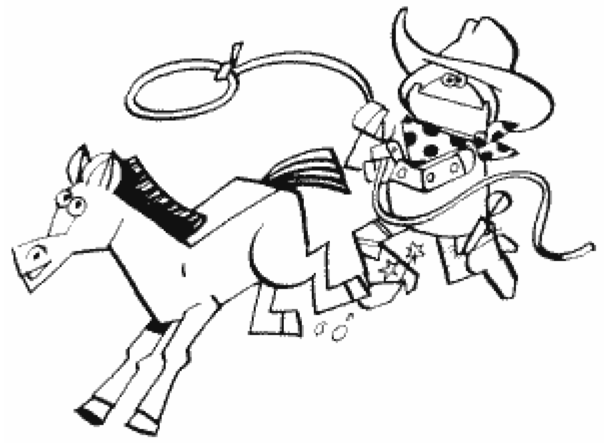
"P"	58416 ($E430)
"C"	58432 ($E440)
"E"	58368 ($E400)
"S"	58384 ($E410)
"K"	58400 ($E420)
"D"	1995 ($07CB)
"R"	varies

FIGURE 26. Addresses for handler address table

The address for "R" varies depending on whether you have a disk drive hooked up and, if so, what kind of DOS you are using. PRINT PEEK(813)+256*PEEK(814) will give you the address for your particular setup.

You can use the preceding addresses to take a look at the handler entry point tables. The addresses in these tables are in the same order that the routines were listed (OPEN, CLOSE, etc.). Don't forget that each address is two bytes long with the exception of the last one, which includes a JMP instruction (76).

HATABS is 38 bytes long, which means there is room for 12 3-byte entries (the last 2 bytes are set to zero and ignored). Even if you are using the disk and RS-232 handlers, that still leaves five entries free. These entries are initially set to zeros, but they're free for your use if you want to write your own handler.



Since the task of writing your own handler is one that most people won't really get into, I'm not going to go into any more detail on it here. If you're interested, De Re Atari and the OS manual should provide all the information you need.

One more thing you'll need to know. CIO searches for a handler address from the end of the table up to the beginning. This means that if you write your own screen handler, for example, CIO will use it instead of the original one.

INPUT/OUTPUT CONTROL BLOCKS (IOCBS)

Back at locations 32 through 47, we ran across something called the Zero-page Input/Output Control Block (ZIOCB). The ZIOCB gets its values from one of the eight Input/Output Control Blocks (IOCBs) located at locations 832 through 959. Basically, the IOCBs are nothing more than a bundle of information used to communicate between the user and the handlers. BASIC usually takes care of them for you in commands like OPEN, PLOT, LPRINT, and so on (all the BASIC I/O commands).

Each IOCB is 16 bytes long, and those bytes are named and used as follows:

ICHID (one byte) This is an offset into HATABS (794 through 831) that points to the name of the device that the IOCB is OPENed for. For example, try the following:

```
100 IOCB1=848:HATABS=794
110 OPEN #1,4,0,"K:"
120 INDEX=PEEK(IOCB1)
130 PRINT "You just OPENed device ";CHR$(PEEK(HATABS+INDEX));":"
```

ICHID is set by the OS.

ICDNO (one byte)
This is the device number. One for D1, two for R2, etc. It is also set by the OS.

ICCOM (one byte)
This is the command that specifies what kind of I/O operation we are going to be doing. See the appendices for a list of possible commands. It is set by the user.

ICSTA (one byte)
This is the status of the last I/O operation. See the OS manual (pp. 165-166) for a list of possible values.

ICBAL/H (two bytes)
This is either the address of the data buffer, or the address of the user's filename (depending on the command).

ICPTL/H (two bytes)
This is the address minus one of the put one-byte routine for the device being used. If the IOCB isn't OPEN, then it points to CIO's error routine for an illegal put.

ICBLL/H (two bytes)
This is the number of bytes that still have to be transferred. Note that under some circumstances not all bytes will be transferred.

ICAX1 (one byte)
This is an auxiliary byte (and is also called **AUX1**), meaning that it helps out ICCOM in specifying what is to be done. It is usually used to modify the OPEN command, but you can use it for your own handlers. With the OPEN command, it is the first value after the IOCB number (#n is the IOCB number), with the bit meanings in Figure 27.

-------1	(1)	Append
------1-	(2)	Directory
-----1--	(4)	Read
----1---	(8)	Write
--1-----	(32)	OPEN screen without erasing screen memory

FIGURE 27. ICAX1 bit meanings

Some combinations are not allowed on some devices. For example, OPEN#1,12,0, "D:TEST" would open a disk file called TEST, and let you read and write to that file. This wouldn't work on a cassette file though.

ICAX2 (one byte)
This is the second auxiliary byte, and is also called **AUX2**. There is no common use for this or any of the other auxiliary bytes; their use depends on the handler. For example, if AUX2 is equal to 128, the cassette handler will put shorter gaps between the records on the tape when it writes data.

ICAX3/4 (two bytes)
These auxiliary bytes are used by BASIC's NOTE and POINT commands to keep track of the sector number. They are not also called AUX3 and AUX4.

ICAX5 (one byte)
This is the fifth auxiliary byte and is also used by NOTE and POINT as the number of the byte within the sector.

ICAX6 (one byte)
OK, enough of the "this is" garbage. I won't even insult your intelligence by telling you it's the sixth auxiliary byte. It has no specific use.

You can use the IOCBs directly by POKEing the values you want into them and then doing a JSR CIOV (58454). See CIOV for more details.

Note that the descriptions for the ZIOCB (32-47) are worded differently from the preceding descriptions, so be sure to read them as well for a better understanding.

IOCB0
832-847   0340-034F

This is, obviously, IOCB zero. If you're using the screen editor, then don't use IOCB0; that's what the screen editor uses. If you are using the screen editor though, you can do neat things by telling the IOCB to send the data somewhere else, like the printer. Try this if you have a printer:

```
100 GRAPHICS 0
110 PRINT "Now we're on the screen"
120 POKE 838,166:POKE 839,238
130 PRINT "now we're on the printer"
140 POKE 838,163:POKE 839,246
150 PRINT "Now we're back on the screen"
```

What we're doing here is changing ICPTL/H to point to the printer's put-one-byte routine rather than the screen editor's. Note that this doesn't turn your computer into a typewriter. The screen editor isn't responsible for putting characters you type on the screen; it only works for things the computer prints on the screen.

Another neat thing you can do to the screen editor is give AUX1(842) a value of 13. This tells it to "append," which doesn't really make any sense. What it does, though, is act as though you were continually pressing the return key. This lets you write a program that will change itself. You simply print some program instructions on the screen, position the cursor on the line of the first instruction, and POKE 842 with 13 to start the computer generating RETURNS, which reads in each line. When the computer POKE's 842 with a 12, the process stops and everything is back to normal. While you do this, the lines of code will appear on the screen, but you can make them invisible by setting the color of the letters to the color of the screen. Try this:

```
10 ? "{CLR}":LIST 30
20 FOR I=1 TO 1000:NEXT I
30 ? "NOW LINE 30 SAYS EVERYTHING ELSE"
40 FOR I=1 TO 1000:NEXT I
50 ? "{CLR}"
60 POSITION 2,10:? "30 ?";CHR$(34);"NOW LINE 30 SAYS
SOMETHING ELSE."
70 POSITION 0,2
80 POKE 842,13
90 POSITION 2,17:? "CONT"
100 POSITION 0,2:STOP
110 POKE 842,12
120 ? "{CLR}":? "NOW LETS SEE WHAT LINE 30 SAYS"
130 ? "THE PROGRAM MODIFIED ITSELF"
140 LIST 30
```

Keep in mind that if you try and delete the lines that change location 842, you'll confuse the heck out of the computer and it will just keep on "pressing" RETURN forever

The screen editor, and therefore lOCB zero, is used in graphics mode zero and in other graphics modes that use text windows. Since lOCB zero is dedicated to the screen editor, however, you should stay away from it even if you're not using the text editor.

IOCB zero is not closed by a NEW, RUN, or LOAD command. All the others are.

IOCB1
848-863 0350-035F

lOCB one.

IOCB2
864-879 0360-036F

IOCB two.

IOCB3
880-895 0370-037F

IOCB three.

IOCB4
896-911 0380-038F

IOCB four.

IOCB5
912-927 0390-039F

IOCB five.

IOCB6
928-943   03A0-03AF

IOCB six. If you're in a mode other than zero, then IOCB six is used for the screen (IOCB0 is used for the text window).

IOCB7
944-959   03B0-03BF

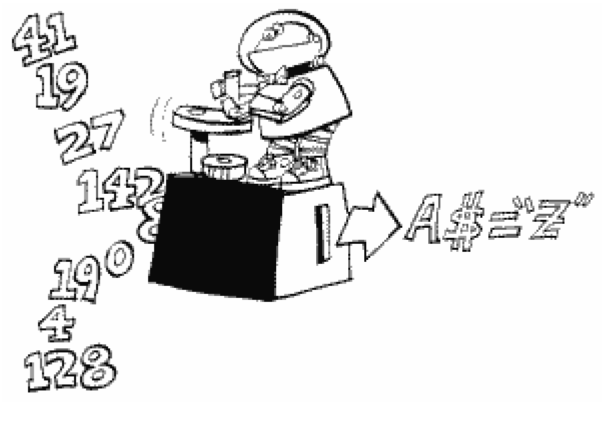
IOCB seven is used by BASIC for I/O to the printer, disk drive, and cassette. That means that this is a pointer to the buffer that will be used to store the data that is to be sent or received during I/O. It's set by the handler to the system buffer at CASBUF (1021) unless you tell the handler differently.

If a GET STATUS command is given, then DBUFLPO/HI is set to point to DVSTAT (746).

If you're communicating with SIO directly, you should make sure you set DBUFLO/HI yourself.

PRNBUF
960-999   03C0-03E7

This is the print buffer, 40 bytes long, used in sending data to a printer. See PBPNT (29; $001C) and PBUFSZ (30; $001E) for details on how this works.



Forty bytes,, as you may be aware, is somewhat shorter than most printer lines (most have 80 character lines). The OS can usually handle this, but sometimes it runs into problems. Semi-colons and commas at the end of LPRINT statements especially tend to mess it up. Several sources briefly mention that the Atari can deal with an 80 column printer if you call it "P2". If this is true, then you would have to OPEN a special IOCB for it and therefore couldn't use LPRINT (you'd have to use PUT and the likes). You're probably better off just to put up with the quirks.

Noname
1000-1020 03E8-03FC

These bytes are marked as being spare, but again, be careful about using them.

CASBUF
1021-1151 03FD-047F

This is the cassette buffer, which is where the cassette handler reads and writes data from and to. It's also used to hold the first disk record when a disk is booted (the OS doesn't know where to put the disk file in memory until it gets a chance to look at this record; see BOOTAD [578,579]).

A cassette record is made up of 132 bytes. Only 128 of these are actual data; the other 4 help out the cassette handler. How? I'm glad you asked. The first 2 bytes, as we learned at CBAUDL/H (750,751), are used to help the handler figure out the correct baud rate. The third byte tells the handler how much data is in the file. It can have the following values:

250 means that there are less than 128 bytes of meaningful data (there will still be 128 bytes, but some of them toward the end will be zeroes). The 128th data byte will give the actual number of meaningful bytes.

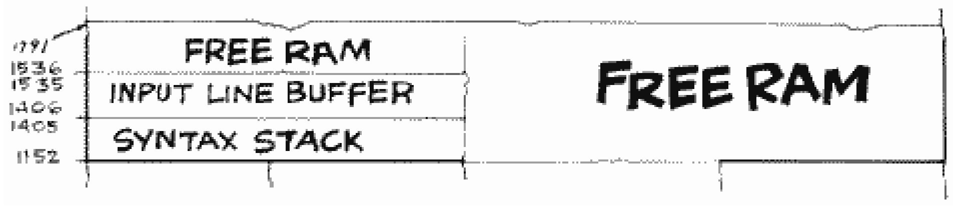
252 means that all 128 bytes are important.

254 means that this is the last record in the file and all 128 bytes will be equal to 0.

The next 128 bytes are the actual data. Notice that they will be stored in CASBUF starting at 1024 and ending at 1151.

But wait, that was only 131 bytes and you said there were 132. Where does the 132nd go if we already filled the buffer? The last byte in a cassette record is the checksum, which is used to make sure that the rest of the data was read correctly. It gets stored at CHKSUM (49), and you should take a look at CHKSUM for a more detailed description of how a checksum works.

Take a look at BPTR (61) and BLIM (650) for more information on the way the buffer is used.



Locations 1152 through 1791 are not used by the OS. Most of them are, however, used by BASIC and/or the floating point package. Only the locations in page six (1536 to 1791) are not used by either. See page six for more information.

SYNSTK
1152-1405 0480-057D

This is BASIC's syntax stack. Unfortunately, since there doesn't seem to be any information on what it's for, I can't explain it to you (I never claimed to be perfect). I suspect, however, that it's used during the tokenization of the BASIC program, since BASIC has the runtime stack (see RUNSTK [142,143]) to use when the program is actually running.

LBPR1
1406 057E

LBUFF prefix one. Again, no information on this one.

LBPR2
1407 057F

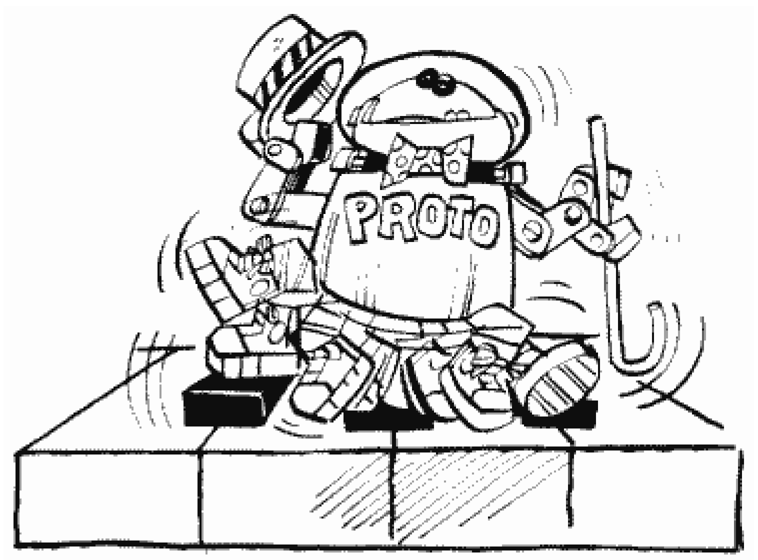
LBUFF prefix two, also not explained anywhere.

LBUFF
1408-1535 0580-05FF

Before I go on, a few words on locations like these. Atari was very nice in releasing the OS listing; a lot of other computer companies don't. Atari did not, however, because of legal restrictions, extend that niceness to BASIC and the floating point package. Therefore, locations that are used by these two are very difficult to explain. The useful locations have been documented, so they can be understood and used by yourself. Ones like these, however, are somewhat obscure, so that you should never have to use them. In other words, don't feel that you're not getting something you'll need.

Now that I've freed myself from the responsibility of properly explaining these locations, I'll actually give you some information on this one. This is a buffer used in converting floating point values to ATASCII values. It's pointed to by INBUFF at locations 243 and 244. Now INBUFF supposedly points to the buffer used to convert ATASCII to floating point, so I suspect that LBUFF swings both ways.

LBUFF is also referred to as the "input line buffer," which implies that this is where a BASIC line is stored when you first type it in.



Location 1535 is the last byte in the buffer and so it is also called LBFEND. Notice that the next three locations are all within LBUFF.

PLYARG
1504-1509 05E0-05E5

Polynomial arguments. Sure!

FPSCR
1510-1515 05E6-05EB

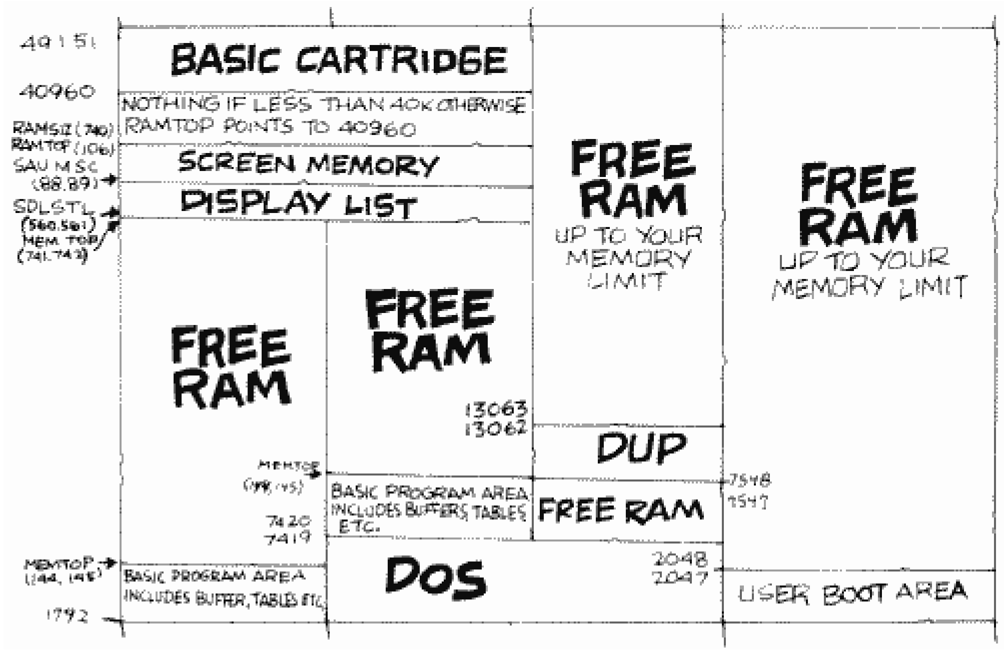
This is like a work area for the floating point package, eh.

FPSCR1
1516-1535 05EC-05FF
The same, only bigger.

PAGE SIX

Locations 1536 through 1791 are normally not used by the OS, BASIC, or the floating point package. That leaves them free for your use (page six is a good place to store a machine language routine). Now I did say that they are “normally” not used. That means that they’re not completely safe. If you try and INPUT more than 128 bytes during I/O, then the extra bytes are stored in page six. That means one of two things. Either don’t INPUT more than 128 bytes at a time, or only use the second half of page six (locations 1664 through 1791). These locations are absolutely guaranteed not to be used by anything no matter what.

Please notice that I only said the OS, BASIC, and the floating point package wouldn’t use page six. If you are using another language, it might, so check the documentation that comes with it.



PAGE SEVEN, EIGHT, NINE, …

If you're not using a disk drive, then location 1792 is the beginning of free memory. If you're using BASIC, "free memory" doesn't mean memory for you to use; it means memory for BASIC to use. There's a difference between the two, and you should go back to locations 128 through 145 if you don't know what it is.

If you are using a disk drive, then the locations from 1792 to the address stored in MEMLO (743,744) are used by DOS and the File Management System (FMS). The value of MEMLO will depend on the version of DOS that you're using, and also on a couple of other things that will be mentioned next. Use PRINT PEEK (743) +PEEK(744)*256 to find out the value for your particular setup.

This book is designed to teach you about your Atari and not about DOS, so I'm not going to go into any detail about how it works or what the memory locations are for. If you're interested in DOS, take a look at COMPUTE's book Inside Atari DOS. I will, however, mention a few useful locations, and give you an idea of how DOS is arranged in memory. This information applies to DOS 2.0S only.

FMS

FMS, or the File Management System, takes up locations 1792 through 5439. What's an FMS? The FMS is essentially the disk handler. The OS does have a disk handler built in, but it doesn't work the same way as the other handlers (see the OS manual for more details) and under normal circumstances is only used to load in the FMS. That means that the disk entry in HATABS (794 through 831) points to the handler information pertaining to FMS.

Page seven (1792 to 2047) is also called the "user boot area" for some reason. "User boot area" implies that this area is free for loading boot files into, which is true. Everything after page seven is also free though, so technically it should also be considered as part of the user boot area. It's only a name though, so do with it what you will.

There is a small bug in the FMS that can cause problems when a file has been OPENed for update (updating adds information to the end of a file rather than erasing it and starting over again). To get rid of it, run the following program and then re-save DOS using menu selection "H":

```
100 FOR LOC=2592 TO 2599
110 READ DAT
120 POKE LOC,DAT
130 NEXT LOC
140 POKE 2625,16
150 POKE 2773,31
160 DATA 130,19,73,12,240,36,106,234
```

SABYTE
1801 0709

In case you didn't know already, DOS limits the number of files you can have OPEN at one time to three. Since you have a total of seven free IOCBs though, there is nothing wrong with having all seven open at one time (other than the fact that DOS tells you you can't). SABYTE lets you change DOS's mind. If you POKE it with some number from one to seven, then that becomes the number of files you can have open at once. You do pay a price, however. For each file that you allow to be open (whether it actually is open or not), a 128-byte data buffer is reserved. So don't make SABYTE larger than you need.

Oh, DOS will only remember this change as long as the computer is turned on. If you want a special version of DOS that always remembers, change SABYTE and then re-save DOS using menu selection "H".

DRVBYT
1802 070A

DOS originally expects to see no more than two disk drives. This can also be changed using DRVBYT as in Figure 28.

0000---1	means that drive one is available.
0000--1-	means that drive two is available.
0000-1--	means that drive three is available.
00001---	means that drive four is available.

FIGURE 28. DRVBYT bit chart

Notice that a value of three in DRVBYTE does not mean that DOS can handle three drives. The binary representation of three is 00000011, which means that drives one and two are available.

Each available drive also has a 128-byte buffer reserved for it, so don't waste

memory by telling DOS you have more drives than you really do (if you only have one drive, you can save 128 bytes by setting DRVBYT to one).

Once you've changed DRVBYT, you can make your change a permanent part of DOS by re-saving DOS using menu selection "H", just like you did for SABYTE.

VERFLG
1913 0779

Actually, this location doesn't have a name so I made one up. In any case, this is the "write verify flag." What is "write verify"? When you save something to disk, DOS first writes the data onto the disk and then reads it back to verify that it was written correctly. This takes time, and the disk very seldom makes an error (very, very seldom). So, if you want to turn the verify off, POKE VERFLG with an 80. To turn it back on again, POKE VERFLG with an 87.

I personally have never had a problem writing without verify. There is, however, a slight chance that the data will not be saved correctly, and if this happens, your program probably won't work. Therefore, you may want to have the verify on when you're saving important programs.

As in the last two locations, menu selection "H" will create a version of DOS that includes your change to VERFLG.

DSKFLE
3118 0C2E

More than once I have ended up with two files on a disk with the same name. If you tell DOS to erase one of the files, both will be done. By changing the normal value here to a zero, DOS will only erase the first file.

WLDCRD
3783 0EC7

Look at your DOS manual and find out what a wildcard is. The wildcard that comes with DOS is the asterisk ("*"). If you want to change it to something else, this is where its ATASCII value is stored.

Again, re-save DOS if you want the change to be permanent.

CHRLO, CHRHI
3818,3822 0EEA,0EEE

DOS only lets you use uppercase letters and numbers in your filenames. This too can be changed. CHRLO and CHRHI define the range of ATASCII characters that can appear in a filename. They are originally set to 65 ("A") and 91 ("["). CHRHI,

as you can see, actually holds the ATASCII value of the character after the last allowable one. The number appear to be taken care of elsewhere and are automatically included in the list of allowable characters.

A few warnings if you decide to change CHRLO and CHRHI. First of all, don't define a space or a period as an allowable character. If you do, you may not be able to load your file back in again. Also, if you include the asterisk as an allowable character, don't forget to change WLDCRD.

To specify uppercase, lowercase, and numbers as allowable characters, set CHRLO to 48 and CHRHI to 123. No guarantees are made if you make CHRHI any larger than this.

Don't forget to re-save DOS so it includes your changes.

Noname
4148,4149 1034,1035

Another problem with DOS is error message #164, which means your data is messed up on the disk. It's fine if you have really goofed up your file, but what if it has only a few problems. Perhaps a few numbers are unreadable. You would like to be able to load these in and fix them on the screen. NO WAY usually. Try POKEing 234 into location 4148 and 234 into 4149 also. These are NOP (no operation) instructions in machine language. Now you will never get error #164. For that reason I don't suggest using this permanently, but rather only when you need it.

Noname
4226 & 4229 1082, 1085

Disk directory sector number. Here is a nice trick discovered by Cordon Banks, a dedicated ATARI nut like those of us here. Did you ever want to have two disk directories? Of course not, but think about it for a moment. A second directory could be used to protect certain files on a disk from being there when others look at the disk. You might have a second set of programs become available to a program after a certain point in the game or drill was passed.

What you do is treat these two locations as the low byte and high byte of the number of the sector that the directory should start on. Normally this is 361, which means 4226 holds 105 and 4229 holds 1 (105+1*256=361). If you wanted to have your directory start at sector 700, POKE in 188 and 2. Now the only problem is how to get the directory to 700. Simply do the POKES before you save the files to the disk. Then POKE the normal values back in before writing DOS to the disk.

DOS

Remember that FMS is just a handler, which means it can only do a limited number of things to the disk. DOS takes care of the more complicated tasks and tells FMS what is can do to help get these tasks done. A large part of DOS, called the Disk Utilities Package (DUP), stays out of memory until you call DOS. The rest can be found in locations 5440 through 6779. This part of DOS, called "mini-DOS," is mainly used to load in DUP from the disk and take care of the MEM. SAV file (see your DOS manual) if necessary.

DOSLO, DOSHI
5446,5460 1546,154A

These two locations hold the address that BASIC will jump to when you call DOS. They initially hold the same value as DOSVEC (10, 11).

BUFFERS

Following the resident part of DOS are the buffers. There are two drive buffers and three data buffers, unless you specify otherwise by changing SABYTE (1801) and/or DRVBYT (1802). The buffers start at 6781 ($1A7D) and, since each buffer takes up 128 bytes, and at 7420 ($1CFC) unless you've made the changes described. This means that MEMLO (743,744) normally points to 7420 if you're using disk.

DISK UTILITIES PACKAGE (DUP)

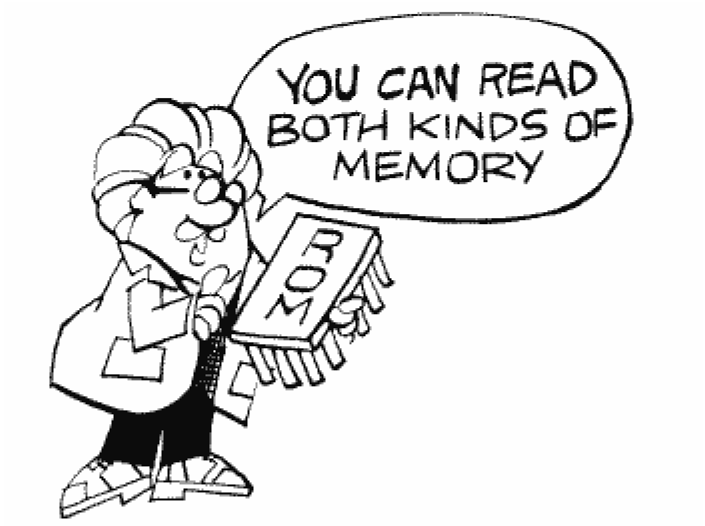
You call DOS from BASIC and "whirrr… beep… beep…" from the disk drive, right? So what's loading in? All the routines that are needed to perform the menu selections that appear on the screen. These routines, along with the menu itself and various buffers and the like, load into locations 7548 through 13062 ($1D7C through $3306). Look at the DOS listing if you need more information about the routines. By the way, if you go to BASIC from DUP and haven't loaded or created a BASIC program that is large enough to wipe out any of the DUP file, you can go to DUP by typing

X=USR (8309)

instead of typing DOS. This saves all the time of loading in DUP again.

FREE RAM

The memory area from the address pointed to by MEMLO (743,744) up to that pointed to by RAMTOP (106) is free RAM. That doesn't mean you didn't pay for it; it means that it is unused. If you are using BASIC, then your program uses the memory from the address pointed to by MEMLO up to the address pointed to by MEMTOP (144,145).

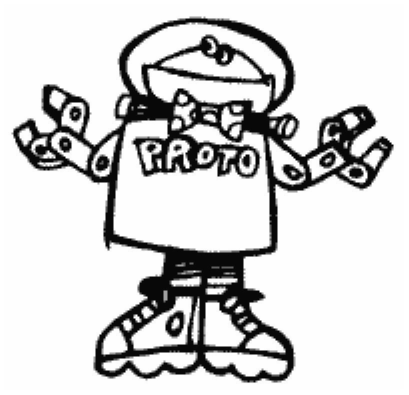


As mentioned already, the value of MEMLO will depend on whether or not you are using disk (and the RS-232 handler, which takes up another 1728 bytes). The value in RAMTOP depends on how much total memory you have. The various values it can have are listed in Figure 29. Don't forget that the value in RAMTOP is the high byte of the address (the number of pages).

MEMORY	RAMTOP	BYTES
8K	32	8192
16K	64	16384
24K	96	24576
32K	128	32768
40K	160	40960*
48K	192	49152*

FIGURE 29. RAMTOP chart

***These values depend on whether or not any cartridges are in place. See the sections on cartridges.**



CARTRIDGE B (RIGHT CARTRIDGE)

The cartridges are a strange breed. They contain their own 8K of ROM, yet they feel the need to shove 8K of RAM out of the way in order to run. The right cartridge gives notice to locations 32768 through 40959 ($8000 through $9FFF). This means that if you have 40K of RAM or more, you'll lose 8K of it. Note that since the 800 is the only Atari that has a right cartridge slot, few companies have cartridges for the right slot.

If a right cartridge is present, TSTDAT at location seven gets set to one during power up.

CARTRIDGE A (LEFT CARTRIDGE)

OK, all Atari computers have a left cartridge slot. Since it may be the only slot, it makes more sense to refer to it as cartridge A.

Cartridge A takes up memory locations 40960 through 49151 ($A000 through $BFFF). This will only affect you if you have 48K of RAM, since these locations are the last 8K.

The last six bytes in a cartridge provide the information that the OS needs in order to run the cartridge. Thus:

49146,49147 ($BFFA, $BFFB) holds the starting (run) address of the cartridge.

49148 ($BFFC) equals zero if a cartridge is plugged in, and doesn't if one isn't.

49149 ($BFFD) tells the OS how to get the cartridge going. If bit zero is set, then the OS boots the disk before it runs the cartridge. If bit two is set, then the cartridge is initialized but not run (if it's not set then it gets run).

49150,49151 ($BFFE,$BFFF) holds the initialization address of the cartridge.

Note that these addresses are all for cartridge A. For cartridge B, just subtract 8192 from each address.

If cartridge A is present, TRAMSZ at location six gets set to one during power up.

If you're using BASIC, then the BASIC cartridge is cartridge A. Because this book is designed to teach you about your Atari, and not about the languages that can be used with it, I'm not going to give you a detailed listing of all the locations in the BASIC cartridge. Don't feel as though you're missing out on something great, however; there's very little in there that would be useful to you. The OS listing does mention four routines, however, so I will mention those.

SIN
48551 BDA7

This routine calculates the sine of the number in floating point register zero (FR0). You should take a look at FR0 (212 to 217) and RADFLG (251) if you're going to try to use it. It might also help to disassemble the code for the routine to get an idea of what's going on.

COS
48561 BDB1

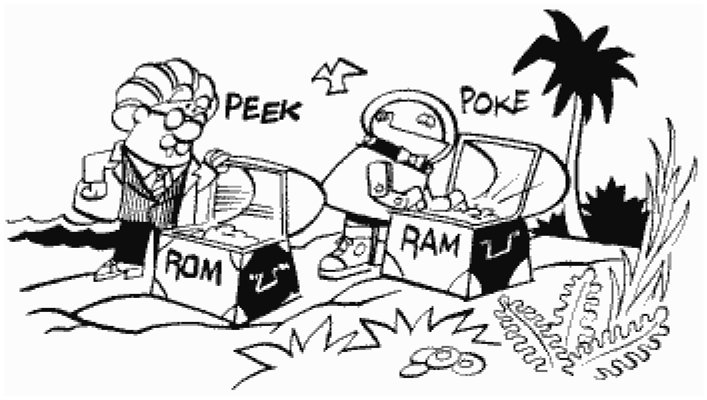
This routine calculates the cosine of FR0.

ATAN
48759 BE77

The arctangent of FR0.

SQR
48869 BEE5

And lastly, the square root of FR0. Note that the carry is significant in all of these operations, in that it will be set if an error occurs during the operation.



SPECIAL CHIPS AND ROM

OK, we’re now done with all the RAM locations. That leaves the ROM and the special chips, which covers the GTIA (or CTIA), POKEY, PIA, and ANTIC chips along with the OS ROM. We’ll learn what each of the chips does as we come across it.

ROM, of course, stands for Read Only Memory, which means that you can’t store values in it. You can’t store values in the chips, either, but as you read through the following locations you’ll notice that I tell you to POKE to them. What’s going on here? When you POKE a value in a chip location, the chip will see the value and act accordingly, but won’t store it anywhere. That means that if you POKE some location with a value and then PEEK that location, you won’t necessarily get the value you POKEd. It’s OK though, because the chip knows what you were trying to do and acts as though the value is in the location. Confused? Just remember that POKE works, but PEEK won’t always.

Because the value you POKE is different from the value you PEEK, a lot of the ROM locations have different meanings depending on whether you’re PEEKing or POKEing. In such a case, I’ll give you an explanation of each.

Now that you think you understand, one more thing to try and trip you up. You can’t POKE to the OS ROM at all. Well, you can if you want to, but nothing will happen.

Let’s do a little memory mathematics. First of all, let’s start with a 48K Atari. It’s actually a 64K Atari. Where’s the other 16K? The OS ROM takes up 10K, so that leaves 6K unaccounted for. 1.25K is taken up by the chips mentioned, so we’re down to 4.75K. Would you believe that 4.75K of memory is unused? Well, you should, because it is. Unfortunately, it’s all in ROM and therefore you can’t use it either. 4K of it is right here, at locations 49152 through 53247 ($C000 through $CFFF). The other .75K, which, by the way, is equal to three pages since a page is .25K, is found amongst the chips at locations 53504 through 53759 ($D100 through $D1FF) and

54784 through 55295 ($D600 through $D7FF). Anyway, that means we have 48K to program with, right? Nope. The BASIC cartridge takes up 8K, as we saw, so we're down to 40K. We already saw that the first 1792 bytes (or 1.75K) are used by BASIC, the OS, and the floating point package, and then on top of that there's screen memory and the memory that the FMS and DOS use if you're using a disk drive. That leaves anywhere from about 37K down to 29K or less! Oh well, I guess even computers can lose their memories (OK, OK, I'm sorry).

GTIA OR CTIA

Time to answer a question that may have been nagging at a lot of you. What's the difference between GTIA and CTIA? And for that matter, what is a CTIA/GTIA? Let's start with the second question first. Way back when the Atari computers were still being designed, they had nicknames. The 400 was called Candy, and the 800 Colleen. So much for trivia. Both Colleen and Candy needed some way of talking to the television set, so a Television Interface Adapter chip was designed. That's where you get the name CTIA from (Colleen/Candy Television Interface Adapter). The CTIA chip gets information from ANTIC and uses it to tell the television set what to do. So much for that.

When the CTIA was originally designed, it was supposed to support 12 graphics modes (0 to 11). Unfortunately, the last 3 modes couldn't be implemented in time to make the production deadline. Rather than hold off releasing the computers, Atari decided to put them out without those three modes and add the modes later. So that's why the early computers don't support modes 9, 10, and 11. After the first batch of CTIAs ran out, Atari starting putting in chips that had the extra 3 modes, for some reason calling them GTIAs, with the "G" standing for George. Maybe George was the guy who finally figured out how to get the last modes to work, I don't know. In any case, that's the difference between CTIA and GTIA. Oh, GTIA

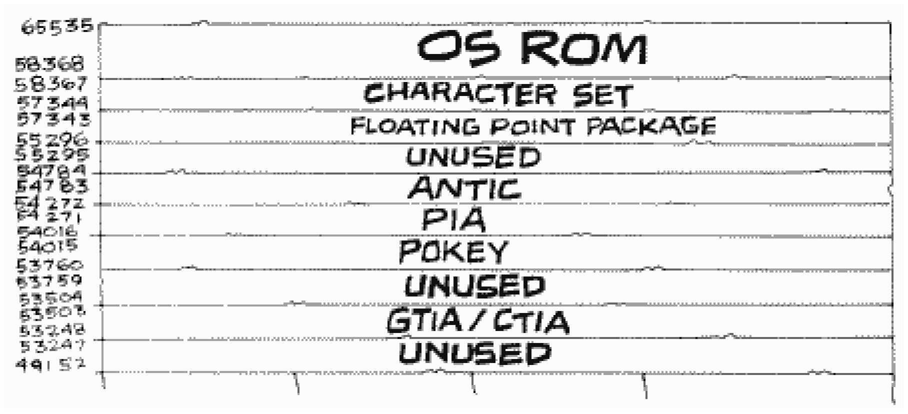
also corrects another problem that CTIA had. It seems that you couldn't line up players and playfield exactly (they were off by half a color clock). To correct this problem, GTIA shifts the playfield by half a color clock. Unfortunately, this means that colors obtained by artifacting (see COLOR1 at location 709 [02C5]) will be the opposite of what they're supposed to be. If you're writing a program that uses artifacting, and you want to make sure that the colors come out right on both the CTIA and the GTIA, there is a program in Appendix Four that you can use to check which chip is present. Once you know that, you can shift everything over one column if necessary.

The CTIA/GTIA takes up locations 53248 through 53505 ($D000 through ($D0FF).

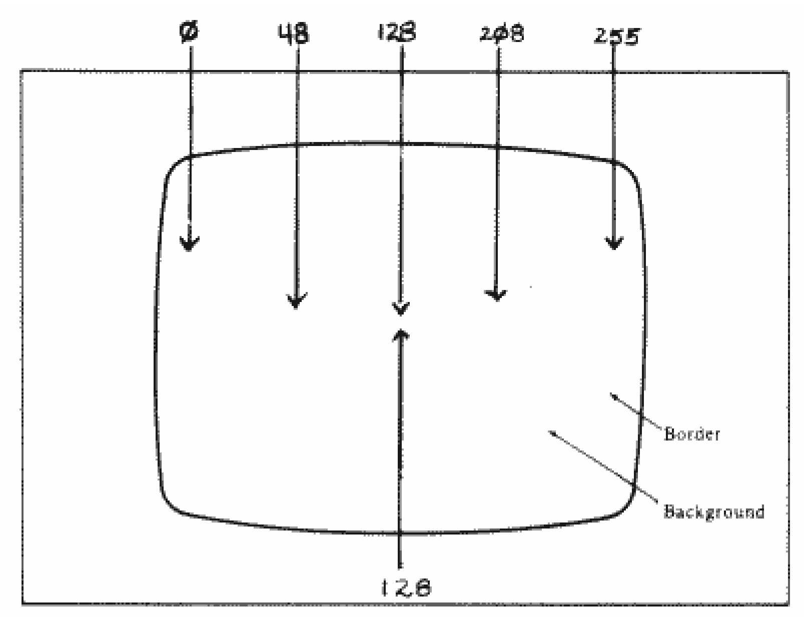
You're now about to enter player/missile territory. That's right, CTIA/GTIA also takes care of player/missile graphics. Since this is such a detailed and popular subject, there is a whole appendix in the back of the book devoted entirely to telling you how to program player/missile graphics. Therefore, the descriptions here will be short and sweet.

Please see the appendix if you don't know what's going on.

HPOSP0 (POKE) and M0PF (PEEK)
53248 D000



(POKE) HPOSP0 specifies the horizontal location of player zero, and can range from 0 to 227. Note that some of these positions will be off the edge of the screen, so experiment to see what values can be seen on your screen. For most television sets, anything between 48 and 208 will be visible.



Because this location is in one of the chips, you will not be able to PEEK here to find out the current position of the player (see the following for what you get when you PEEK). That means that you should keep track of the position in a separate variable.

(PEEK) M0PF is a collision register. Collision registers are used to tell who's collided with whom on the screen (in other words, who's sharing the same pixels with whom). They can be very useful in games and other applications. Note that the collision registers do not work properly in GTIA modes nine through eleven. You should also check HITCLR at location 53278.

M0PF tells you what part of the playfield missile zero has collided with (the background is not considered part of the playfield here). Its bits have the meanings in Figure 30.

-------1	(1)	means a collision with playfield zero.
------1-	(2)	means a collision with playfield one.
-----1--	(3)	means a collision with playfield two.
----1---	(4)	means a collision with playfield three.

FIGURE 30. M0PF bit chart

HPOSP1 (POKE) and M1PF (PEEK)
53249 D001

(POKE) HPOSP1 specifies the horizontal position of player one.

(PEEK) M1PF is the collision register for missile one/playfield collisions.

HPOSP2 (POKE) and M2PF (PEEK)
53250 D002

(POKE) HPOSP2 specifies the horizontal position of player two.

(PEEK) M2PF is the collision register for missile two/playfield collisions.

HPOSP3 (POKE) and M3PF (PEEK)
53251 D003

(POKE) HPOSP3 specifies the horizontal position of player three.

(PEEK) M3PF is the collision register for missile three/playfield collisions.

HPOSM0 (POKE) and P0PF (PEEK)
53252 D004

(POKE) HPOSM0 specifies the horizontal position of missile zero (missiles move just like players).

(PEEK) P0PF is the collision register for player zero/playfield collisions. It works just like the missile/ playfield collision registers.

HPOSM1 (POKE) and P1PF (PEEK)
53253 D005

(POKE) HPOSM1 specifies the horizontal position of missile one.

(PEEK) P1PF is the collision register for player one/playfield collisions.

HPOSM2 (POKE) and P2PF (PEEK)
53254 D006

(POKE) HPOSM2 specifies the horizontal position of missile two.
(PEEK) P2PF is the collision register for player two/playfield collisions.

HPOSM3 (POKE) and P3PF (PEEK)
53255 D007

(POKE) HPOSM3 specifies the horizontal position of missile three.

(PEEK) P3PF is the collision register for player three/playfield collisions.

SIZEP0 (POKE) and M0PL (PEEK)
53256 D008

(POKE) SIZEP0. You can set the size of each player (in terms of width) to one of three possibilities, each of which is twice as wide as the one before it. A value of zero or two in this or the next three locations specifies normal width, which is eight color clocks wide (the same width as a graphics mode two character). Similarly, a one specifies double width, and a three quadruple width.

A player is normally eight bits wide. Changing the width does not affect this but rather shows each bit two or four times in a row (for example, Figure 31).

Normal Width (zero)

For Player 0: POKE 53256,0

```
10011001
10111101
11111111
10111101
10011001

#  ##  #
# #### #
########
# #### #
#  ##  #
```

FIGURE 31 (partial). Changing player widths

Two things to note here. First, the height of the player is not changed, and second, each player can be set separately.

Double Width (one)

POKE 53256,1

```
1100001111000011
1100111111110011
1111111111111111
1100111111110011
1100001111000011

##    ####    ##
##  ########  ##
################
##  ########  ##
##    ####    ##
```

Quadruple Width (three)

POKE 53256,3

```
1111000000001111111100000000111
1111000011111111111111110000111
1111111111111111111111111111111
1111000011111111111111110000111
1111000000001111111100000000111

####        ########        ####
####    ################    ####
################################
####    ################    ####
####        ########        ####
```

FIGURE 31 (continued). Changing player widths

(PEEK) M0PL is the collision register for missile zero/player collisions. Its bits have the meanings in Figure 32.

-------1	(1)	means a collision with player zero.
------1-	(2)	means a collision with player one.
-----1--	(3)	means a collision with player two.
----1---	(4)	means a collision with player three.

FIGURE 32. M0PL bit chart

Because of the way player/missile graphics is designed, there is no way to detect a collision between two missiles.

SIZEP1 (POKE) and M1PL (PEEK)
53257 D009

(POKE) SIZEP1 specifies the width of player one.

(PEEK) M1PL is the collision register for missile one/player collisions.

SIZEP2 (POKE) and M2PL (PEEK)
53258 D00A

(POKE) SIZEP2 specifies the width of player two.

(PEEK) M2PL is the collision register for missile two/player collisions.

SIZEP3 (POKE) and M3PL (PEEK)
53259 D00B

(POKE) SIZEP3 specifies the width of player three.

(PEEK) M3PL is the collision register for missile three/player collisions.

SIZEM (POKE) and P0PL (PEEK)
53260 D00C

(POKE) The missiles only have one location that is used to specify their widths, and SIZEM is it. This is how the bits are used (Figure 33).

------00 or ------10 ------01 ------11	(0) (2) (1) (3)	 for normal width missile zero. for double width missile zero. for quadruple width missile zero
----00-- or ----10-- ----01-- ----11--	(0) (8) (4) (12)	 for normal width missile one. for double width missile one. for quadruple width missile one
--00---- or --10---- --01---- --11----	(0) (32) (16) (48)	 for normal width missile two. for double width missile two. for quadruple width missile two.
00------ or 10------ 01------ 11------	(0) (128) (64) (192)	 for normal width missile three. for double width missile three. for quadruple width missile three.

FIGURE 33. SIZEM bit chart

(PEEK) P0PL. Now it's time for player-to-player collision registers. P0PL has exactly the same meaning as M0PL, except it detects player zero/player collisions rather than missile zero/player collisions. Note that a player will never collide with itself.

GRAFP0 (POKE) and P1PL (PEEK)
53261 D00D

(POKE) When G/CTIA is drawing the screen, it relies on the five GRAF registers (this one and the four following) to tell it what the players and missiles will look like (GRAF stands for "GRAFics register"). Every time G/CTIA comes to a horizontal position on the screen where a player/missile is supposed to be, it looks at the corresponding register to see what to put on the screen. Now it's probably occurred to you that the GRAF registers are only one byte long. That means that unless someone puts new values in them every time a new line on the screen gets drawn, all the players and missiles will just look like a bunch of vertical lines. So who's going to do the changing? If you're adventurous (and good with machine language), you can do it yourself, but there's a much easier way. If we tell GRACTL (53277) and DMACTL (54272) to turn on DMA (Direct Memory Access) for players and missiles, then ANTIC will very thoughtfully keep changing the GRAF registers for us. All we have to do is put a description in memory of how we want the players and missiles to look. See PMBASE (54279) for information on how to do that.

If you want a quick way to create some kind of vertical border, use the GRAF registers without DMA. By POKEing a value of 255 into GRAFP0, for example, you can get a vertical band the height of the screen. Note, however, that you have to turn off DMA for all players.

(PEEK) P1PL is the collision register for player one/player collisions.

GRAFP1 (POKE) and P2PL (PEEK)
53262 D00E

(POKE) GRAFP1 is the graphics register for player one.

(PEEK) P2PL is the collision register for player two/player collisions.

GRAFP2 (POKE) and P3PL (PEEK)
53263 D00F

(POKE) GRAFP2 is the graphics register for player two.

(PEEK) P3PL is the collision register for player three/player collisions.

GRAFP3 (POKE) and TRIG0 (PEEK)
53264 D010

(POKE) GRAFP3 is the graphics register for player three.

(PEEK) Remember the STRIGs back at locations 644 through 647? Well, they were the shadow registers for the TRIGs, which work the same way. A zero means the joystick button is pressed (joystick zero in this case), a one means it isn’t. With the TRIGs, you can make it so that if a button is pressed, TRIG will stay set to zero until you reset it. See GRACTL (53277) to find out how. Otherwise, TRIG will return to one as soon as the button is released.

For those of you who are hardware minded, the TRIGs read the value of pin six of the controller jacks.

GRAFM (POKE) and TRIG1 (PEEK)
53265 D011

(POKE) GRAFM is the graphics register for the missiles. Since missiles are only two bits wide, all four can fit into one register. The bits are assigned as in Figure 34.

BIT	7	6	5	4	3	2	1	0
USE	M	3	M	2	M	1	M	0

FIGURE 34. GRAFM bit chart

If all the missiles are in the same byte, how do you change one without affecting the others? It's not easy from BASIC. You have to somehow break the byte up into four parts, change the part you want, and then put it back together. See the appendix on player/missile graphics for a routine to help you do this. In machine language, all you have to do is use AND to look at the bits you want and to clear them, and ORA to reset them. Don't forget that you can't PEEK GRAFM, so you'll have to keep track of its value in a separate variable (which is taken care of for you if you're using DMA). Again, see the appendix for more of an explanation.

(PEEK) TRIG1 is the value of joystick button one. It has a shadow register STRIG1 at location 645 ($285).

COLPM0 (POKE) and TRIG2 (PEEK)
53266 D012

(POKE) This specifies the color and brightness of player zero and missile zero. Sound familiar? That's because it has a shadow register called PCOLR0 at location 704. Look there for more information.

(PEEK) TRIG2 is the value of joystick button two. It has a shadow register STRIG2 at location 646 ($286).

COLPM1 (POKE) and TRIG3 (PEEK)
53267 D013

(POKE) COLPM1 is the color and brightness of player one and missile one. It has a shadow register PCOLR1 at location 705.

(PEEK) TRIG3 is the value of joystick button three. It has a shadow register STRIG3 at location 647 ($287).

COLPM2 (POKE) and PAL (PEEK)
53268 D014

(POKE) COLPM2 is the color and brightness of player two and missile two. It has a shadow register PCOLR2 at location 706.

(PEEK) It would be too easy to make fun of this location, so I'll spare you. What a PAL, huh? There are three types of television standards around the world: PAL, NTSC, and SECAM (a television standard just specifies how the television should work). PAL is used in Europe, NTSC here in North America, and SECAM in France, Russia, and parts of Africa. Atari has two different versions of the computer, one for PAL and one for NTSC (so we'll ignore SECAM). What's the difference? The PAL Ataris run about 25 percent faster than the NTSC Ataris. Also, PAL televisions have more scan lines than NTSC ones, which means that VBLANK occurs approximately every fiftieth of a second rather than every sixtieth.

"So what?" you're saying (I can hear you). Actually, unless you're programming for a PAL Atari, nothing I just said will be of any concern to you. If you are programming for a PAL, however, you should be aware of the differences.

I almost forgot. PAL (the memory location) is used to determine what kind of Atari you have. The bits have the following meaning:

----000- (0) means that the computer is set up for PAL.
----111- (14) means that it's set up for NTSC.

Make sure you only check bits two, three, and four (i.e., on an NTSC computer, PEEK(53268) won't necessarily give you 14; it might give you 15).

COLPM3 (POKE only)
53269 D015

COLPM3 is the color and brightness of player three and missile three. It has a shadow register PCOLR3 at location 707.

COLPF0 (POKE only)
53270 D016

OK, we're out of the player/missile color registers and into the playfield color registers. See the shadows at locations 708 through 712 for more information.

COLPF0 is the color and brightness of playfield zero and has a shadow register COLOR0 at location 708.

COLPF1 (POKE only)
53271 D017

COLPF1 is the color and brightness of playfield one and has a shadow register at location 709.

COLPF2 (POKE only)
53272 D018

COLPF2 is the color and brightness of playfield two and has a shadow register at location 710.

COLPF3 (POKE only)
53273 D019

COLPF3 is the color and brightness of playfield three and has a shadow register at location 711.

COLBK (POKE only)
53274 D01A

COLBK is the color and brightness of the background and has a shadow register at location 712.

PRIOR (POKE only)
53275 D01B

PRIOR is used to select the priorities of the various objects on the screen. What? Check its shadow register GPRIOR at location 623 for an explanation.

VDELAY (POKE only)
53276 D01C

As you'll find out later, at location DMACTL (54272), you can choose one- or two-line resolution for your players and missiles. If you choose two-line, however, you have to move things vertically two lines at a time rather than one. VDELAY can be used to move things down by one line. Its bits specify the object to be moved (Figure 35).

-------1	(1)	means move missile zero down one.
------1-	(2)	means move missile one.
-----1--	(4)	means move missile two.
----1---	(8)	means move missile three.
---1----	(16)	means move player zero.
--1-----	(32)	means move player one.
-1------	(64)	means move player two.
1-------	(128)	means move player three.

FIGURE 35. VDELAY bit chart

To continuously move an object down, just repeat the following loop:

1. Set the appropriate bit to one.

2. Move the shape data forward one byte in memory and set the bit back to zero.

3. Go to step 1.

To continuously move it up:

1. Move the data back one byte and set the appropriate bit to one.

2. Set the bit back to zero.

3. Go to step 1.

GRACTL (POKE only)
53277 D01D

When the computer is first turned on, it is not set up to accept player/missile data from ANTIC (player/missile DMA is turned off). Before you can get G/CTIA and ANTIC to talk to each other about players and missiles, you have to tell G/CTIA to accept player/missile data and ANTIC to send it. GRACTL is used to do the former

(DMACTL [54272] is used for the latter). Here's the way the bits are used (Figure 36).

-------0	(0)	tells G/CTIA not to accept missile data.
-------1	(1)	tells it to accept missile data.
------0-	(0)	tells it not to accept player data.
------1-	(2)	tells it to accept player data.

FIGURE 36. GRACTL bit chart

What about turning players and missiles off? Can you just POKE GRACTL with a zero? Not always; POKEing GRACTL will stop new data from coming in but will not get rid of whatever's in the GRAF registers (see the preceding). That means you have to turn off GRACTL (and DMACTL) and then POKE all the GRAF registers with zeroes. If you're doing this from BASIC, you may want to POKE all the horizontal position registers with a zero first, thus moving everybody off the edge of the screen. That way, you won't have to look at any garbage that may appear in them before you get a chance to change the GRAFs.

GRACTL is also used to make a slight change to the joystick and paddle buttons. Normally, these buttons will give a value of zero when they are pressed, and a value of one when they're not. Sometimes, however, they may get pressed and released before you get a chance to check them. GRACTL helps you solve this problem as in Figure 37.

-----1--	(4)	will "latch the trigger inputs."
-----0--	(0)	will clear the trigger inputs.

FIGURE 37. GRACTL/Joystick bit chart

Whoa! Wanna translate that to English? Well, "trigger input" is just a fancy term for button value. "Latch" means that once you press a button, its value will stay zero until you clear the trigger inputs, even if the button is released before then. Ta da!

Note that you cannot latch or clear individual button values.

The joystick button values are stored in the preceding TRIG locations, and the paddle button values are in PORTA at location 54016.

HITCLR (POKE only)
53278 D01E

After you've looked at the collision registers and seen who has run into whom, it's a good idea to set them to zero. Why? If you don't, then the old collision values will still be there the next time you check, and you won't know whether anything new has happened or not. HITCLR to the rescue. POKE any value into HITCLR and all the collision registers will be cleared (set to zero).

CONSOL
53279 D01F

(PEEK) We're going to do this one backwards (PEEK before POKE), and you'll see why in a second. PEEKing CONSOL will tell you which of the buttons on the keyboard (OPTION, SELECT, and START) are pressed. The bits are used as in Figure 38.

-------0	(0)	means that START is pressed.
-------1	(1)	means it isn't.
------0-	(0)	means that SELECT is pressed.
------1-	(2)	means it isn't.
-----0--	(0)	means that OPTION is pressed.
-----1--	(4)	means it isn't.

FIGURE 38. CONSOL bit chart

(POKE) If you're looking at CONSOL from machine language, you have to first POKE it with an eight. This tells it to clear out the old values and bring in the new. Don't worry about it in BASIC.

If you POKE CONSOL with a value less than eight, the speaker inside your Atari will make a clicking sound. If you keep POKEing it, you'll get a buzz. Try this:

```
100 POKE 53279,0:GOTO 100
```

Why do you have to keep POKEing the zero over and over? Because the OS automatically stores a value of eight in CONSOL every stage two VBLANK, which turns off the speaker.

You can make changes to the preceding line to get slightly different buzzes. For example,

```
100 POKE 53279,0
110 POKE 53279,8:POKE 53279,8
120 GOTO 100
```

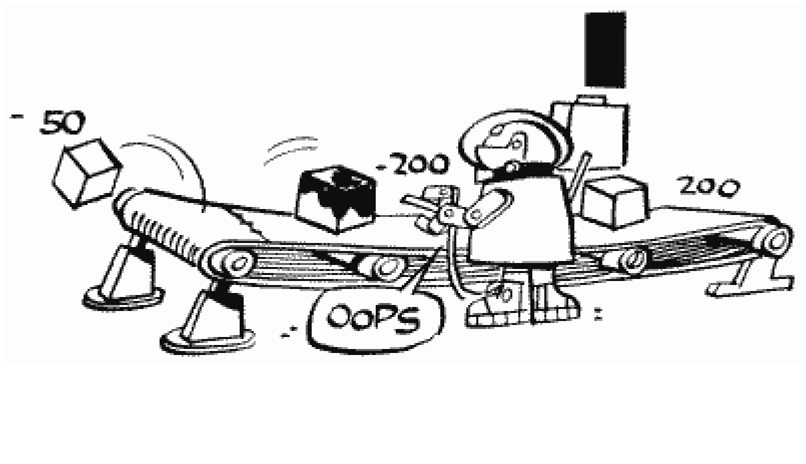
That's all there is to the GTIA/CTIA chip. Even though it takes up another 224 bytes (through location 53503), they are not used at the moment. Could that mean incredible additions in the future that will allow you to do three-dimensional movement with hundreds of different colors and incredibly realistic detail? Nah!

POKEY

POKEY handles a whole bunch of stuff, including sound, the paddles, the keyboard, IRQ interrupts, and serial I/O, so let's get right into it.

SOUND

BASIC, with its simple SOUND statements, doesn't even hint at the complex options available from your Atari for sound generation. That's a shame, since sound can be as important a part of your program as graphics. If you're interested in learning more about how the sound capabilities of your computer work, everything you need to know appears next. If you really want to apply these capabilities to complicated sound effects, I urge you to get your hands on a copy of De Re Atari. It has the most in-depth look at sound and how to program it that I've seen.



TIMERS

Hold on, what happened to sound? Don't worry, I haven't forgotten it; the system timers mentioned back at locations 528 through 533 share a few of the sound locations so I thought we'd get them out of the way first. Because these timers are only going to be used by those of you programming in machine language, I'll just give you a checklist of what you need to do to use them. For more information, consult good old De Re Atari.

1. Use AUDCTL to pick the particular clock frequency you want to use (the system timers count clock pulses).

2. Depending on which timer you're using, use AUDC1, AUDC2, or AUDC4 to set the volume for the associated audio channel to zero.

3. Similarly, set AUDF1, AUDF2, or AUDF4 to the number of pulses you want to count.

4. Make sure your interrupt routine is set up and ready to go.

5. Change VTIMR1, VTIMR2, or VTIMR4 (these start at location 528) to point to your interrupt routine. Note that VTIMR4 will not work in the original version of the OS.

6. Use IRQEN and POKMSK (53774 and 16) to enable the timer interrupts.

7. Lastly, get the timers going by writing any value to STIMER (53769).

You should be aware that DMA, DLIs, and vertical blank processing can affect the timers' performance, so don't rely on complete accuracy.

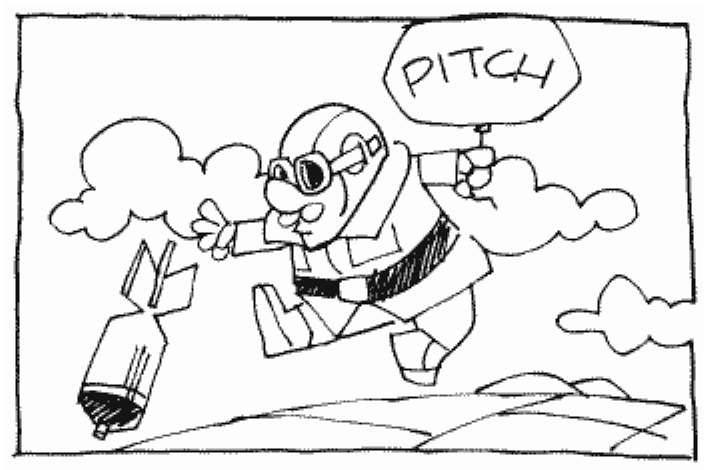
AUDF1 (POKE) and POT0 (PEEK)
53760 D200

(POKE) The BASIC SOUND statement has the following format:

SOUND VOICE, PITCH, DISTORTION, VOLUME

VOICE is a value from zero to three. Think of the Atari as having four separate sound-effects performers inside, called voice zero, voice one, voice two, and voice three. Actually, while BASIC numbers them zero through three, POKEY prefers the more normal one through four. We'll go with POKEY, which also calls them "audio channels" rather than voices. We'll go with it on that, too.

AUDF1 has nothing to do with audio channel numbers other than the fact that it specifies the pitch value for audio channel one, so we'll start discussing pitch. Pitch is another word for frequency, or note, or tone, whichever you prefer. We'll use frequency. Basically, frequency describes how "high" or "low" a sound is (you may prefer the terms "treble" and "bass"). Before we look at how this got the name frequency, we need to understand what makes a sound.



Sound of any type is nothing more than air moving in "waves." What's an air wave (uh-oh, those two words together sound familiar)? It's essentially the same as an ocean wave in that it consists of a large movement of air followed by almost no movement. A whole bunch of waves one after the other make up a sound, and the length of the waves, or how far apart in time they are, determines the frequency of the sound. Do you see now why it's called frequency? Because the frequency of the sound depends on how frequent the waves are.

How does the computer create sound waves? Actually, the speaker creates the waves; the computer just controls the speaker. A speaker makes waves by moving in and out at the desired frequency (each move in and out creates one wave). To make the speaker move in and out, the computer sends it a "pulse" of electricity. A pulse is just a short burst, so this is equivalent to quickly flicking a switch to the speaker on and then off again. This moves the speaker out and then in, thereby creating the air movement we need. All this work just for one little air wave!

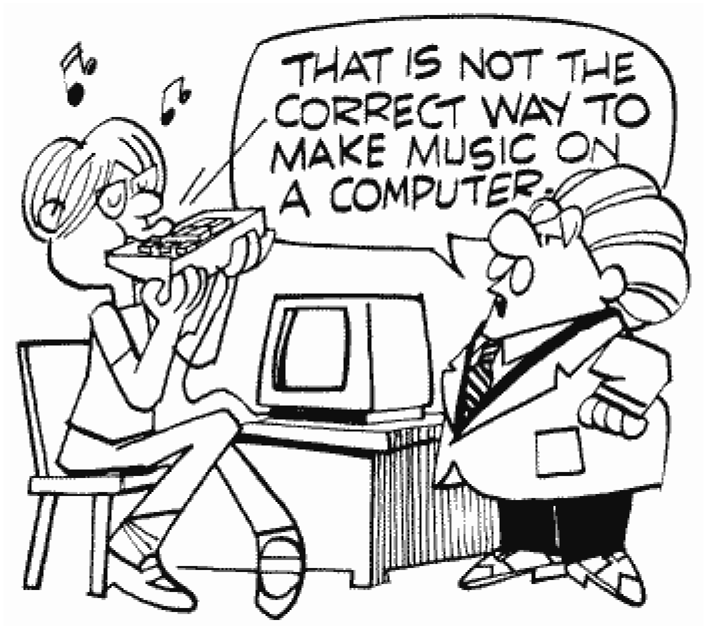
Our next step is to look at where the pulses come from. The frequency of the pulses will determine the frequency of the sound, so it's safe to assume that the AUDF registers will have something to do with it. Try the following statements:

```
SOUND 0,10,10,8
SOUND 0,200,10,8
```

Uh-oh, the higher frequency value gave us a lower frequency sound. What gives? It turns out that the AUDF registers are used to determine the frequency rather than used as the frequency. In technical terms, they specify "N" in divide-by-N circuits. What does that mean in non-technical terms? The Atari computers have several "clocks" on board. These clocks don't tell the time, but rather send out a stream of pulses with a specific frequency. The clock that is usually used by the AUDF registers has a frequency of 64 KHz (Kilohertz, meaning 1,000 pulses per second). A divide-by-N circuit takes this stream as input, and for every N pulses coming in it

sends one out, in this case to the speaker. For example, if N were equal to two, then one pulse would go out for every two coming in, resulting in an output frequency of 32 KHz. If N were equal to 128, the output would have a frequency of 0.5 MHz, or 500 Hz (hertz, meaning one pulse per second). Now you should be able to see where the "divide-by-N" name comes from.

There is one important detail that I failed to mention. In the case of Atari sound, before the pulses from the divide-by-N circuits get to the speaker, they automatically go through a divide-by-two circuit, regardless of what the value of N was. This means that our two examples would result in actual sound frequencies of 16 KHz and 250 Hz. You should also note that POKEY adds one to the value you put in AUDF before it gives it to the divide-by-N circuit. That results in a possible frequency range of 125 Hz (64 KHz/256/2) to 32 KHz (64 KHz/1/2). Most human ears can't hear sounds higher than 20 KHz, so there's more than enough in the high-frequency range. In the low range, you can hear down as far as 20 Hz, so you can see that the low end is lacking. We'll see ways a little later on that allow you to get around that.



(PEEK) POT0 is the value of paddle zero. Since it has a shadow register PADDL0 back at location 624, you should go back there for a description.

Machine language programmers should also consult ALLPOT at location 53768 and POTGO at location 53771.

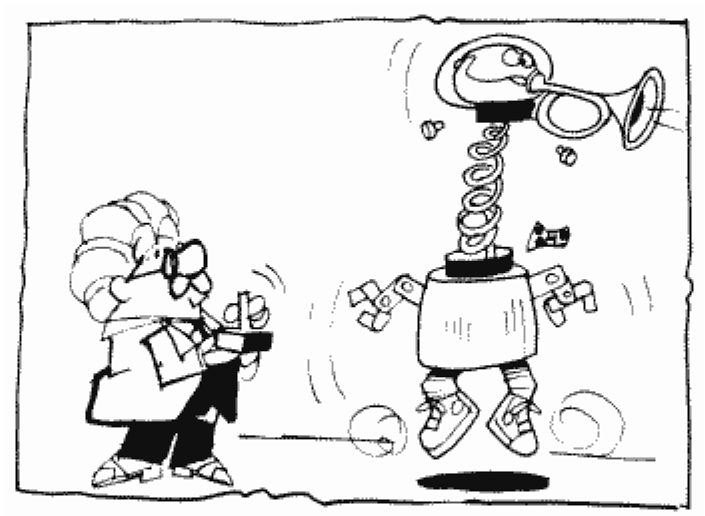
AUDC1 (POKE) and POT1 (PEEK)
53761 D201

(POKE) The four AUDC registers are used to specify the DISTORTION and VOLUME parts of the SOUND statement (see AUDF1). Their official name is the "audio control registers." Bits zero through three are used for volume, and bits four through seven for distortion. (actually five through seven. Bit four is "volume only")Here's how the volume bits work (Figure 39).

----0000	(0)	means no volume (no sound)
----0001	(1)	means the lowest volume
.		
.		
.		
----1111	(15)	means the highest volume

FIGURE 39. AUDC1 bit chart

How does volume tie in to our discussion on sound waves? The higher the volume, the more electricity there is in each pulse to the speaker. The more electricity there is in each pulse, the greater the distance the speaker moves in and out. The greater the distance the speaker moves, the larger the sound wave. Finally, the larger the wave, the louder the sound. (Note that "large" refers to height, not length.)



Distortion, unfortunately, is not quite as easy to explain as volume. Let's start by explaining the easiest distortion bit, number four. When bit four is set to one (which adds 16 to the previous value), you control the speaker directly. In other words, AUDF is ignored, and the value in the volume bits sent directly to the speaker. Try the following:

POKE 53761,24

This sends a volume value of eight to the speaker, and you hear a "pop" as a result. Now try it again. Don't bother turning your volume up; there wasn't a pop this time. Why? As long as bit four is set, the volume will always be sent; the speaker will receive a constant stream of electricity rather than a pulse. This moves it out but not back in again. Turn the volume off to move it back in:

POKE 53761,16

You should hear another pop as it moves back. That means that a pulse is actually two pops, but since they happen so close together, the result is one loud pop. Try this:

POKE 53761,24:POKE 53761,16

Now you hear the result of a complete pulse. In any case, this is how you can define your own pulses. BASIC is too slow to really do anything with this technique (called "volume only" sound), but with machine language you define your own frequencies, wave shapes, and sound envelopes, creating anything from the sound of a piano to the sound of a human voice. If this sounds interesting, please see De Re Atari for more details on each of these topics.

We're now left with the other three distortion bits. They involve something called "poly-counters." Without going into explicit detail, a poly-counter takes a stream of pulses and "randomly" removes some of them. This results in a frequency that, although close to the original, is constantly changing by small amounts. This in turn results in a messy sound, often called "noise."

The word "random" is used loosely in the preceding description, because the so-called random pattern will eventually repeat itself. I'm not going to explain why, because the inner-most workings of a poly-counter are not important to us here. See De Re Atari if you're curious. For now, suffice it to say that there are three different types of poly-counters in the Atari; a 17-bit one, a 5-bit one, and a 4-bit one. The greater the number of bits the poly-counter uses, the longer it will take for the pattern to repeat. Let's jump a little ahead of ourselves by listening to the result of the 17-bit poly-counter:

SOUND 0,100,8,8

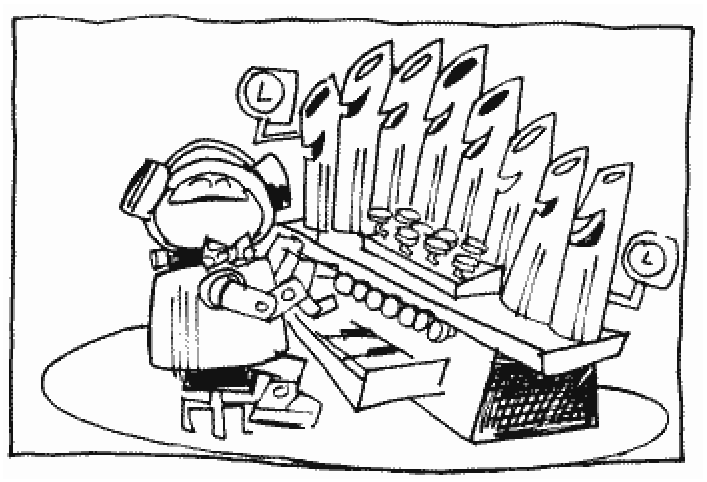
And now the 5-bit:

SOUND 0,100,2,8

And finally the 4-bit:

SOUND 0,100,12,8

Notice how the 4- and 5-bit poly-counters create more of a repetitious sound, while the 17-bit sound is much more random (it seems to be just noise).



By this time it should be apparent that the last three bits are used to select which poly-counters are applied to our unsuspecting frequency. Here are their exact uses (Figure 40).

--0-----	(0)	means that either the four bit or seventeen bit counter will be applied (depending on the value of bit six).
--1-----	(32)	means that neither will be applied regardless of what bit six is set to.
-0------	(0)	means that the seventeen bit counter will be applied if bit five is not set.
-1------	(64)	means that the four bit counter will be applied if bit five is not set.
0-------	(0)	means that the five bit counter will be applied.
1-------	(128)	means that the five bit counter will not be applied.

FIGURE 40. Poly-counters bit chart

You can see that the five-bit counter can be combined with either or neither of the other two. Also note that the divide-by-two circuit that was mentioned under AUDF1 is applied after the poly-counters. In case all of this has you confused, here's all the possible bit combinations and the corresponding order of things (Figure 41).

000-----	(0)	1. divide-by-N circuit 2. five bit poly-counter 3. seventeen bit poly-counter 4. divide-by-two circuit
0-1-----	(32)	1. divide-by-N circuit 2. five bit poly-counter 3. divide-by-two circuit
010-----	(64)	1. divide-by-N circuit 2. five bit poly-counter 3. four bit poly-counter 4. divide-by-two counter
100-----	(128)	1. divide-by-N circuit 2. seventeen bit poly-counter 3. divide-by-two circuit
1-1-----	(160)	1. divide-by-N circuit 2. divide-by-two circuit
110-----	(192)	1. divide-by-N circuit 2. four bit poly-counter 3. divide-by-two circuit

FIGURE 41. Bit combinations for AUDC1

Remember to divide these values by 16 to get the DISTORTION value for BASIC's SOUND command.

After all of this, do you know what I'm going to tell you now? Don't worry about any of it. Everything I just told you is for the sole purpose of letting you understand what's going on behind the sounds you are creating. As long as it sounds good, don't worry what poly-counters are being used. It just doesn't matter.

(PEEK) POT1 is the value of paddle one. It has a shadow register PADDL1 at location 625.

AUDF2 (POKE) and POT2 (PEEK)
53762 D202

(POKE) AUDF2 specifies the frequency for audio channel two.

(PEEK) POT2 is the value of paddle two. It has a shadow register PADDL2 at location 626.

FREE GIVE AWAY!

On the disk or tape that we offer with this book, you will find a nice menu of special sound effects to give you ideas using the sound registers.

AUDC2 (POKE) and POT3 (PEEK)
53763 D203

(POKE) AUDC2 specifies distortion and volume for audio channel two.

(PEEK) POT3 is the value of paddle three. It has a shadow register PADDL3 at location 627.

AUDF3 (POKE) and POT4 (PEEK)
53764 D204

(POKE) AUDF3 specifies the frequency for audio channel three.

(PEEK) POT4 is the value of paddle four. It has a shadow register PADDL4 at location 628.

AUDC3 (POKE) and POT5 (PEEK)
53765 D205

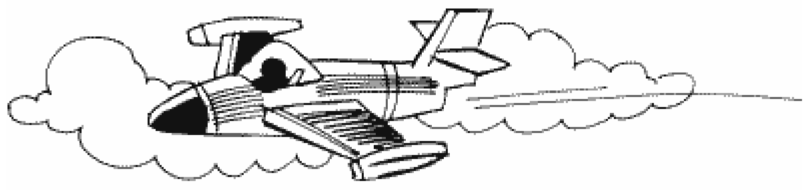
(POKE) AUDC3 specifies distortion and volume for audio channel three.

(PEEK) POT5 is the value of paddle five. It has a shadow register PADDL5 at location 629.

AUDF4 (POKE) and POT6 (PEEK)
53766 D206

(POKE) AUDF4 specifies the frequency for audio channel four.

(PEEK) POT6 is the value of paddle six. It has a shadow register PADDL6 at location 630.



AUDC4 (POKE) and POT7 (PEEK)
53767 D207

(POKE) AUDC4 specifies distortion and volume for audio channel four.

(PEEK) POT7 is the value of paddle seven. It has a shadow register PADDL7 at location 631.

AUDCTL (POKE) and ALLPOT (PEEK)
53768 D208

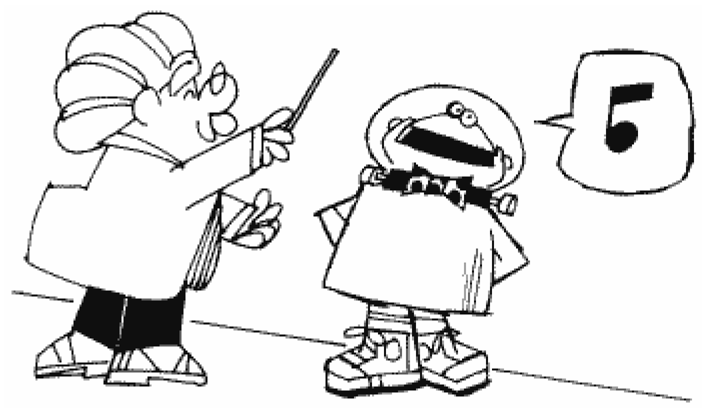
(POKE) AUDCTL is the audio control register. This means that you can use it to make changes to the basic sound setup. So, without any further ado, let's take a look at how its bits are used (unless otherwise noted, a bit set to zero simply cancels the effect of it being set to one) (See Figure 42).

-------0	(0)	means that the 64 KHz clock is the main source of pulses for all channels (unless otherwise specified).
-------1	(1)	means that the 15 KHz clock is used instead.
------1-	(2)	inserts a high-pass filter into channel two and clocks it with channel four.
-----1--	(4)	inserts a high-pass filter into channel one and clocks it with channel three.
----1---	(8)	joins channel four to channel three ("N" becomes sixteen bits long).
---1----	(16)	joins channel two to channel one. (16-bit)
--1-----	(32)	means use the 1.79 MHz clock for channel three (MHz stands for Mega-Hertz, which is one million pulses per second)
-1------	(64)	means use the 1.79 MHz clock for channel one.
1-------	(128)	changes the seventeen bit poly-counter into a nine bit poly-counter.

FIGURE 42. AUDCTL/ALLPOT bit chart

Some of these are probably giving you problems, so let's take a closer look. First of all, we just discussed poly-counters, so there should be no problem there. You should also understand the effect that using different types of clocks will have. The higher the frequency of the clock, the higher the frequency of the sound. For example, try the following statement:

```
SOUND 0,100,10,8
```



Now use bit zero of AUDCTL to change the main clock from 64 KHz to 15 KHz:

POKE 53768,1

Notice how the sound got lower? Now change the clock for channel one to 1.79 MHz:

POKE 53768,64

By using this ability to change the clocks, you can extend the range of frequencies that the Atari can produce. When you use the 1.79MHz clock, however, all the sound will be up in the high range, since even with AUDF set to 255, the resulting frequency will still be 3.5 KHz. How do we get down into the lower frequencies? This is where being able to join two channels together comes in handy. By having sixteen bits available for "N" in our divide-by-N circuit rather than only eight, we can fully utilize the 1.79-MHz clock. Now we can go all the way down to 14 Hz! Try the following program to get the idea. It joins channel two with channel one (channel two is the high byte, channel one the low), switches channel one's clock to 1.79 MHz, and then lets you use joystick zero to change the values of channel one (move the joystick up or down) and channel two (move the joystick left and right):

```
100 SOUND 0,0,0,0
110 POKE 53768,80
120 POKE 53761,160
130 POKE 53763,168
140 CH1=0:CH2=0
150 POKE 53760,CH1
160 POKE 53762,CH2
170 S0=STICK(0)
180 CH1=CH1-(S0=14)+(S0=13)
190 CH1=CH1-CH1*(CH1=256)+256*(CH1=-1)
200 CH2=CH2-(S0=7)+(S0=11)
210 CH2=CH2-CH2*(CH2=256)+256*(CH2=-1)
220 GOTO 150
```

Note that line 120 sets channel one's volume to zero, since channel two is where the sound will come from. Line 130 sets channel two to distortion 10 (pure tone) and volume eight.

The last thing we need to explain is probably what was confusing you most: high-pass filters. Actually, high-pass filters are a relatively simple concept. All they do is stop a frequency from getting through to the speaker unless it's higher than some other specified frequency. For example, let us suppose that we set bit two of AUDCTL. This, we are told, means that a high-pass filter will be inserted in channel one, and channel three will be used to clock it. In English, channel one will only be heard if its frequency is greater than that of channel three. Let's look at a picture of this (Figure 43).

```
    | = = = = = = = = /
    | = = = = = = = /
f   | = = = = = = /
r   | = = = = = /
e   | = = = = /
q   | = = = /
    | = = /
    |/
    ----------------
```

time

FIGURE 43. Frequency vs. time

If the diagonal line represents the sound from channel three, then the shaded area above it represents the frequencies that channel one can play at any given time during that sound. Unfortunately, this isn't quite the way things actually work.

On the Atari, a high-pass filter apparently works by looking at the pulses from each channel. If two pulses coincide, then the high-pass filter will only allow one through. I'm not exactly sure whether one is given priority over the other (it would make sense to give the pulse from the higher frequency priority), but the result is that channel one is not completely cut off when it should be, and is partially cut off when it shouldn't be. Try the following to see what I mean:

```
100 SOUND 0,0,0,0
110 POKE 53768,4
120 POKE 53761,168
130 POKE 53765,168
140 POKE 53760,200
150 POKE 53764,100
160 GOTO 160
```

What you're hearing is a bizarre combination of the pulses coming from channels one and three. According to the definition of a high pass filter, though, you shouldn't be hearing channel one at all, since its frequency is lower than that of channel three. Oh well, at least it allows for some neat sound effects.

(PEEK) ALLPOT is used to determine whether or not the POT value for a particular paddle is valid (see the previous eight locations). If bit n of ALLPOT is set, then POTn contains a valid value for paddle n. This should not concern you unless you're programming in machine language.

STIMER (POKE) and KBCODE (PEEK)
53769 D209

(POKE) POKEing any value in STIMER will get the system timers (POKEY timers) going. See the TIMER section for more information.

(PEEK) When a key is pressed, this location is the first to know about it. From here it goes into the shadow register CH at location 764, which is where you should look for more details.

SKREST (POKE) and RANDOM (PEEK)
53770 D20A

(POKE) POKEing any value here sets bits five through seven of the serial port status register SKSTAT (53775) to zero.

(PEEK) If you do any machine language programming, you've probably wondered at one time or another how to get random numbers. Wonder no more; RANDOM holds the highest 8 bits of the 17-bit poly-counter mentioned under AUDC1. In other words, it will give you a "random" number between 0 and 255. The quotes come from the fact that the values in RANDOM will eventually start repeating themselves (i.e., you'll get the same series of numbers all over again). For all practical purposes, however, you can consider RANDOM to be random.

POTGO (POKE only)
53771 D20B

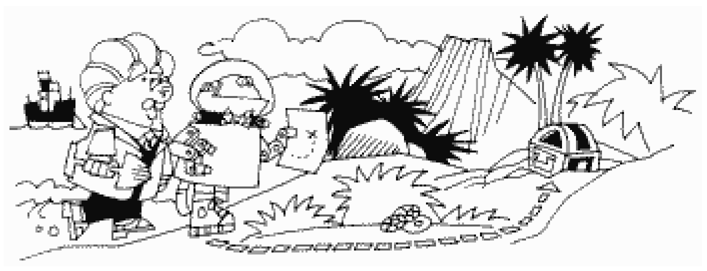
POKEing any value here starts the POT scan sequence. The POT scan sequence is simply the routine that figures out what values should be in the POT registers. The stage two VBLANK routine automatically takes care of POTGO, and you should generally let it. If you decide for any reason to take on the paddles yourself, consult the OS manual and the hardware manual. It's not necessary for most people, so I won't discuss it any further here.

Noname
53772 D20C

This location is not used. Don't forget that it's in the middle of a chip, so don't try to use it yourself.

SEROUT (POKE) and SERIN (PEEK)
53773 D20D

(POKE) SEROUT is used when the serial port needs another byte to send. It's called the eight-bit parallel holding register, a long name that simply means that it holds the byte until the serial output shift register needs it. The serial output shift register then sends out the byte one bit at a time.



SEROUT is usually written to in response to a "serial output data needed" interrupt, which is generated when the serial output shift register needs another byte. See IRQEN.

(PEEK) SERIN is also the parallel holding register, but is used when reading rather than writing.

You usually read the parallel holding register when a "serial input data ready" interrupt occurs. This happens when all eight bits of the incoming byte have been received and transferred to the parallel holding register. See IRQEN.

IRQEN (POKE) and IRQST (PEEK)
53774 D20E

(POKE) IRQEN is used to enable or disable IRQ interrupts. See its shadow register POKMSK at location 16 for a complete description. For more information on interrupts in general, see the section right before location 512.

(PEEK) IRQST is the IRQ interrupt status register. Its bits correspond to the same interrupts as those in IRQEN and are set to zero when the corresponding interrupt occurs. In order to reset it after an interrupt does occur, you must clear the interrupt bit in IRQEN (you can then reset it if you want the interrupt to be able to happen again).

There are two IRQ interrupts that are enabled and have status registers elsewhere (in PIA). See PACTL and PBCTL at locations 54018 and 54019.

SKCTL (POKE) and SKSTAT (PEEK)
53775 D20F

(POKE) At the shadow register for SKCTL (SSKCTL, 562), I shied away from giving a description. I won't do that here, but be forewarned that most of it will be for the expert.

SKCTL controls the configuration of the serial port, determines the type of pot scan to be used, and enables the keyboard circuits. Its bits have the meanings in Figure 44 (unless noted otherwise, a bit set to zero has the opposite effect of when it's set to one).

-------1	(1)	enables the keyboard debounce circuits.
------1-	(2)	enables the keyboard scanning circuit.
-----0--		means that POKEY will take 228 scan lines (one frame) to determine the POT values.
-----1--	(4)	means that POKEY will only take two scan lines to determine the POT values, but they won't be as accurate.
----0---		means that serial output will be transmitted as a logic true/false signal.
----1---	(8)	means that serial output will be transmitted as a two-tone signal (used for cassette data).
-001----	(16)	
. . .		
-111----	(112)	these three bits determine how to transmit and receive data (with respect to clock rates). See page II.27 of the Hardware Manual for a complete description.
1-------	(128)	forces the serial output to zero (forces a break).

FIGURE 44. SKCTL (POKE) bit chart

(PEEK) SKSTAT is a status register for the serial port and the keyboard. Instead of trying to explain, let me just give you the bit meanings (bits are normally set and have the following meanings when they aren't).

If any of the last three bits get cleared, they should be reset to one using SKRES at 53770.

SKSTAT is also helpful if you want to add a voice track to your program. You probably already know that you can play a tape through the TV speaker while you are running a program (see PACTL at location 54018 if not). Unfortunately, if you want what's playing on the tape to coincide with what's going on on the screen, it's difficult to get the timing right. You can, however, put a digital track alongside the voice/music that will tell the computer when to do things. SKSTAT is used in such cases to look at the digital track.

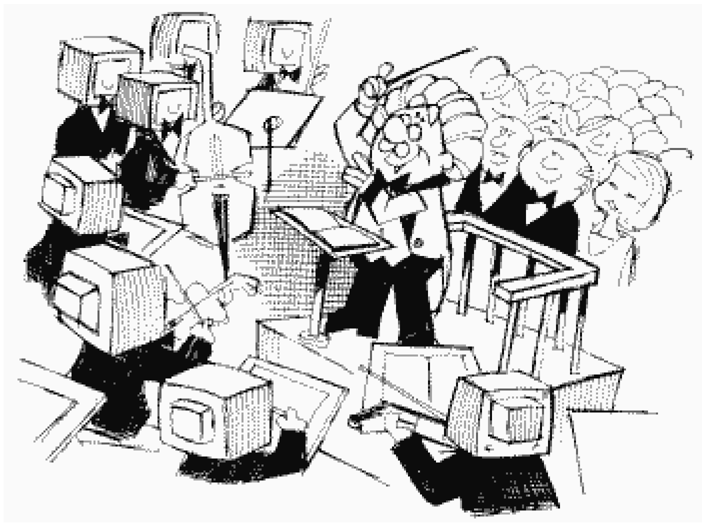
Since this technique has limited applications, I won't go into it here. If you're interested, however, you should take a look at the section on cassette in De Re Atari. It has an excellent explanation, along with the programs needed to both read and write the digital track.

-------1	this bit is not used (and is always set to one.)
------0-	means that the serial input shift register is busy.
-----0--	means that the last key pressed is still pressed.
----0---	means that the shift key is pressed.
---0----	means that you can ignore the shift register and read data straight from the serial input port.
--0-----	means that a keyboard over-run has occurred. To tell you the honest truth, I have no idea what that means.
-0------	means that a serial data input over-run has occurred. Ditto.
0-------	means that a serial data input frame error has occurred.

FIGURE 45. SKSTAT (PEEK) bit chart

Noname
53776-54015   D210-D2FF

These locations are unused at this time, even though they are a part of POKEY.

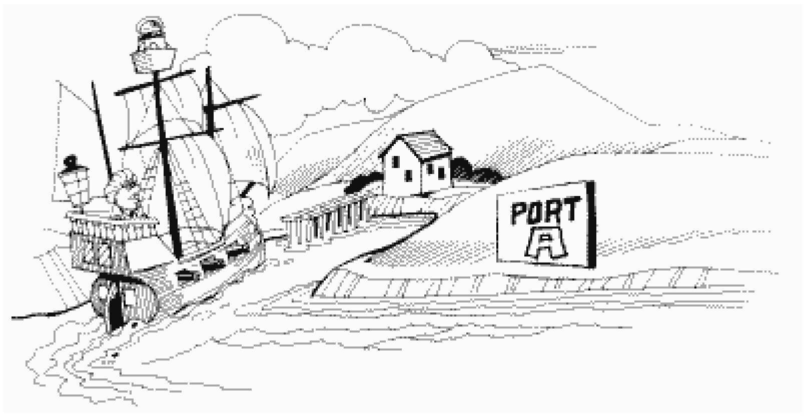


PIA (6520)

PIA stands for Peripheral Interface Adapter and is also known as the 6520 chip. It takes care of the four controller jacks (two on some Atari models), which are the places that you plug your joysticks into. These controller jacks, or Atari ports as we will call them, have capabilities far greater than the reading of joysticks, paddles, and light pens. They can handle simultaneous input and output, which makes them perfect for use as an alternative to the 850 expansion interface. Some companies, in fact, already manufacture a cable that lets you run a printer from these ports instead of the 850.

PORTA
54016 D300

PORTA has two functions actually, depending on whether bit two of PACTL (following) is set. If it is set, then PORTA writes to or reads from the first two controller jacks. Depending on whether you're using joysticks or paddles, PORTA's bits will have the following meanings in Figure 46.



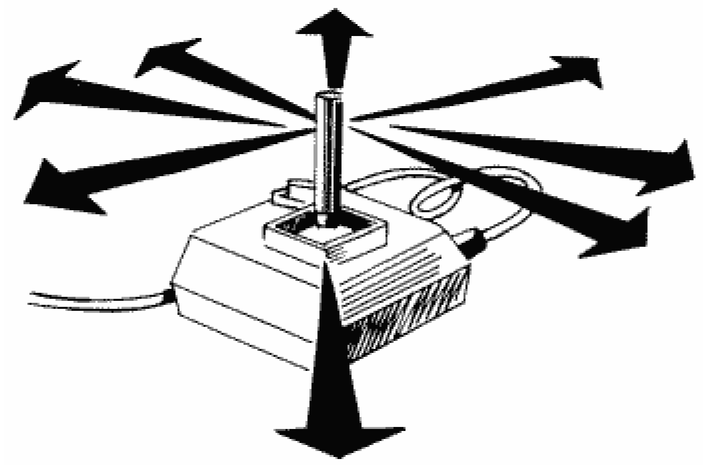
JOYSTICKS	
-------0	means that joystick zero is moved up.
------0-	means that joystick zero is moved down.
-----0--	means that joystick zero is moved left.
----0---	means that joystick zero is moved right.
---0----	means that joystick one is moved up.
--0-----	means that joystick one is moved down.
-0------	means that joystick one is moved left.
0-------	means that joystick one is moved right.

PADDLES	
-----0--	means that paddle zero's button is pressed.
----0---	means that paddle ones' button is pressed.
-0------	means that paddle two's button is pressed.
0-------	means that paddle three's button is pressed.

FIGURE 46. PORTA (paddles/joystick) bit chart

Substitute "is not" for "is" in the preceding descriptions if the bit is set to one.

The shadow registers for PORTA in this sense are STICK0 and STICK1 (632 and 633), and PTRIG0 through PTRIG3 (636 through 639).

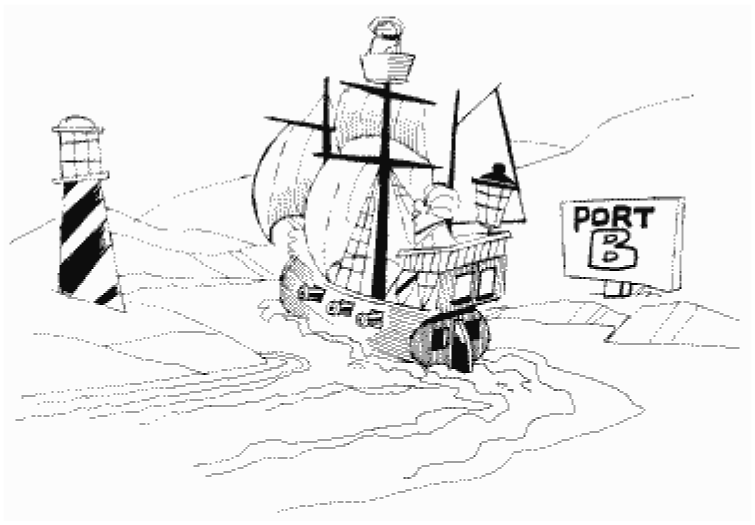


If bit two of PACTL is set, then PORTA writes to the direction control register. A direction control register, as the name implies, is used to specify the direction that information (data) is traveling on the various port pins. What are port pins? Take a close look at the controller ports and you can see them. They are numbered one through five on the top row (left to right) and six through nine on the bottom. PORTA only deals with one through four; bits zero through three represent pins one through four on jack one, while bits four through seven represent pins one through four on jack two. When bit two of PACTL is set, a bit set to one in PORTA means that the corresponding pin will be used for output. Similarly, a bit set to zero means that pin will be used for input.

You may be wondering what the other port pins are used for. Pin five is used for the right paddle value (POT1/3/5/7), pin six for the joystick button (TRIG0/1/2/3), pin seven supplies five volts to the paddles (this pin isn't connected in the joysticks), pin eight is the ground (for both joystick and paddle), and pin nine is the left paddle value POT0/2/4/6).

PORTB
54017 D301

PORTB is the same as PORTA, except it's used with controller jacks three and four rather than two and three. Also, its function is determined by PBCTL, not PACTL.



The shadow registers for PORTB and STICK2 and STICK3 (634 and 635), and PTRIG4 through PTRIG7 (640 through 643).

PACTL
54018 D302

If you have a cassette player hooked up to your computer, try putting a music cassette in it, pressing PLAY, and then entering the following statement:

```
POKE 54018,52
```

This turns on the cassette motor and lets you play music or voice through the TV speaker. To turn off the motor, use the following:

```
POKE 54018,60
```

Other uses of PACTL, which is also called the "port B controller," are as in Figure 47.

-011--00	(48)	disables peripheral A interrupts.
0-011--01	(49)	enables “peripheral proceed line available” interrupts (IRQ, vectored through VINTER at locations 514 and 515).
-011-00-	(48)	means that PORTA above will write to the direction control register.
-011-10-	(52)	means that PORTA will read and write to the first two controller jacks.
-0110-0-	(48)	turns the cassette motor on (also called peripheral motor).
-0111-0-	(56)	turns the cassette motor off.
1011--0-	(176)	means that a “peripheral proceed line available” interrupt has occurred. You cannot write to this bit, but rather clear it by PEEKing PORTA.

FIGURE 47. PACTL (Port A controller) bit chart

A few words on the preceding values before we move on. PACTL is initialized to 60, which means that the cassette motor is turned off (bit three) and PORTA will read and write to the first two controller jacks (bit two). This piece of information should hopefully clear up a question you might have had concerning turning the cassette motor on and off. The reason the POKEs I gave you earlier are different than the bit values above is that bit two should be on in order for the joysticks and paddles to work properly. That’s why we use 52 and 60 in the POKES instead of 48 and 56.

PBCTL
54019 D303

This is the port B controller and has the same functions as PACTL with the following differences:
1. Bits zero and seven deal with the “peripheral interrupt line available” interrupts.

2. Bit two deals with PORTB.

3. Bit three no longer controls the cassette motor but is instead used for peripheral command identification. It is not clear anywhere as to exactly what this means (although most sources also label it as the “serial bus command line”). It is initialized to one.

POKEing 54019 with a 56 tells the computer to take the next POKE to 54017 as a data direction code. This is binary code with each bit corresponding to a pin on the jacks. If the corresponding bit is 1, the pin is defined as output, and if it is 0, then the pin is input. Once you have completed that section of code, you may POKE to 54017 whatever you may want to send out. If you POKE there and then PEEK the same location, you will get back the code you sent, as if it were a RAM location. This means that if you sent the low order four bits as output, and the upper four bits as input, you can send a code out, then read the input combined with the code you sent. This makes scanning the controllers simple to set up in your software. The value you read is what you sent plus 16 times the value that your device sends back.

Here is an example setting up the B port as output:

```
100 POKE 54019,56
110 POKE 54017,255
120 POKE 54019,60
```

The 56 tells the ATARI that the next POKE to 54017 will be a direction control code. It is in binary, so the 255 sets up all eight pins for output. The 60 is then sent out.

Noname
54020-54271   D304-D3FF

Here we are at the end of the useful PIA locations, once again faced with lots of unused locations, as in all of these.

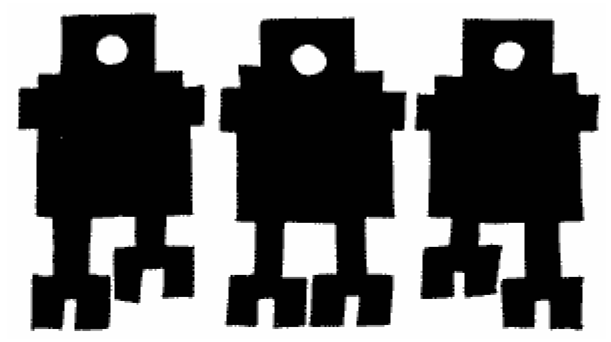
ANTIC

We found out previously that the GTIA/CTIA chip converts information about the screen into a form that the television set can understand. It gets most of this information from ANTIC, which in turn gets it from you.

ANTIC is like a computer within a computer. It has its own special program, called the display list, which in turn has its own special commands. These commands tell ANTIC such things as how the screen is supposed to look and where to find the data that is to appear on it. But we already know this from our discussion at SDLSTL (560,561). ANTIC also takes care of the Non-Maskable Interrupts (NMIs), fine scrolling, and various pointers, all of which will affect the way the screen will appear. Let's take a look.

DMACTL (POKE only)
54272 D400

DMACTL controls DMA (Direct Memory Access). Since there is a wonderful description of it at its shadow register, SDMCTL (559), I won't repeat myself here. There are, however, two more things to add. First of all, DMACTL must be used along with GRACTL (53277) when turning on players and missiles. Secondly, both DMACTL and GRACTL are initialized to 34.



CHACTL (POKE only)
54273 D401

CHACTL makes various changes to the way inverse characters appear and also allows you to make all characters appear upside down (what fun!). See its shadow register CHACT at location 755 for a complete description.

DLISTL, DLISTH (POKE only)
54274,54275   D402,D403

DLISTL/H specifies the address of the beginning of the display list. See its shadow register, SDLSTL, at locations 560 and 561.

HSCROL (POKE only)
54276 D404

Fine scrolling is by far one of the most impressive features the Atari has to offer. We've all been impressed by games that have smoothly scrolling playfields, and wondered, no doubt, how we could do it ourselves. HSCROL and VSCROL are the way, but unfortunately, machine language is required to get the kind of effects you've seen. Although fine scrolling can be done from BASIC, it is not nearly as smooth as machine language and is darn near impossible when you're scrolling more than one or two lines. So, although I'll cover the basics here, if you really want to learn to do great fine scrolling from machine language, check out the excellent section on scrolling in De Re Atari.

HSCROL allows you to fine scroll horizontally, one color clock at a time (a color clock is the size of a graphics mode seven pixel). It will affect every mode line that has bit four set in the corresponding display list instruction (see SDLSTL at locations 560 and 561). For example,

```
100 GRAPHICS 0
110 DLIST=PEEK(560)+PEEK(561)*256
120 POKE DLIST+7,18
130 LIST
140 FOR COLCLK=0 TO 15
150 POKE 54276,COLCLK
160 FOR DELAY=1 TO 50
170 NEXT DELAY
180 NEXT COLCLK
190 GOTO 140
```

As you can see, there are a few things acting screwy here. First of all, why are the lines below the one being scrolled messed up? ANTIC expects to see 48 bytes per horizontally scrolling line instead of the regular 40. In our example, that causes the lower lines to get shifted over. The solution, and the reason that fine scrolling from BASIC is so difficult, is to give each horizontally scrolling line an LMS instruction (again, see SDLSTL). This brings us to our second problem. Why aren't the characters in our example being scrolled all the way across the screen? HSCROL can only handle values between 0 and 15. If you want to scroll more than 15 color clocks, what you have to do is set HSCROL back to 0 and change the LMS addresses of the lines you're scrolling. In graphics mode zero, for example, you would subtract four from each LMS address. Why four? Each character in graphics mode zero is 4 color clocks wide, so the equivalent of setting HSCROL to 16 would be moving the characters four to the right, which is the same as subtracting four from the LMS addresses. As I said before, this can get messy from BASIC.

Let's look at a checklist of what you need to do to have fine horizontal scrolling:

1. LMS addresses for all the lines you are going to scroll.

2. Bit four set on all the display list instructions for the lines you are going to scroll.

3. Screen memory set up properly to account for the longer lines.

And to do the actual scrolling:

1. Set HSCROL to 0 (15 if you're scrolling from right to left).

2. Add one to HSCROL (subtract 1). Remember that HSCROL is POKE only, so you'll have to keep track of its current value in a separate variable.

3. If HSCROL equals 16 (minus 1), set it to 0 (15) and subtract (add) 4 to each LMS address. If you're using graphics modes one or two, add or subtract 2 instead of 4, since each character in these modes is twice as wide as in graphics zero.

4. Go to step 1.

When you change the LMS addresses, you should also check to make sure that you haven't scrolled too far to the left or right.

SCROLLING IS LIKE HAVING A MOVING SIDEWALK GOING BY WITH TEXT AND GRAPHICS ON IT.
HOTEL
PROTO

VSCROL (POKE only)
54277 D405

VSCROL is like HSCROL, except it takes care of fine vertical scrolling. Bit five in the display list instructions is responsible for turning the scrolling on or off for each line (one equals on), and the value you POKE into VSCROL is the number of scan lines you want to scroll the lines upwards. Try the following:

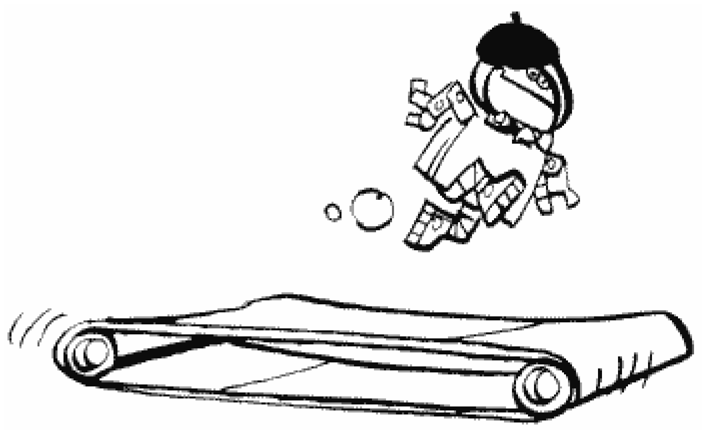
```
100 GRAPHICS 0
110 DLIST=PEEK(560)+PEEK(561)*256
120 POKE DLIST+7,34
130 LIST
140 FOR SCNLIN=0 TO 7
150 POKE 54277,SCNLIN
160 FOR DELAY=1 TO 50
170 NEXT DELAY
180 NEXT SCNLIN
190 GOTO 140
```

There are a few things to notice here. First of all, there's no problem with the lines below the one scrolling. Vertical scrolling does not expect extra bytes per line, so there is no need to worry about that. Secondly, try BREAKing the program and

then POKE 54277,0.

Where's line 120? In fine vertical scrolling, the line after the last line to be scrolled acts as a "buffer." The buffer provides data to scroll into the last scrolling line. To see this, get rid of line 190 and RUN the program again. You should always make sure that there is one nonscrolling line to act as the buffer.

Our final thing to notice here is that every now and then the screen "jumps" a little while the program is running. This is because ANTIC does not like you changing VSCROL (or HSCROL) while it's trying to draw the screen. That means that you should take care of fine scrolling during VBLANK, which is another reason why machine language is necessary.



One nice thing about vertical scrolling is that you only need to have an LMS instruction on the first line to be scrolled. For example, try the following:

```
100 GRAPHICS 0
110 DLIST=PEEK(560)+PEEK(561)*256
120 POKE DLIST+3,98
130 FOR INSTR=6 TO 27
140 POKE DLIST+INSTR,34
150 NEXT INSTR
160 LIST
170 LMSLO=PEEK(DLIST+4)
180 LMSHI=PEEK(DLIST+5)
190 FOR SCNLIN=0 TO 7
200 POKE 54277,SCNLIN
210 FOR DELAY=1 TO 50
220 NEXT DELAY
230 NEXT SCNLIN
240 LMSLO=LMSLO+40
250 IF LMSLO>255 THEN LMSLO=LMSLO-256:LMSHI=LMSHI+1:POKE
    DLIST+5,LMSHI
```

```
260 POKE DLIST+4,LMSLO
270 GOTO 190
```

This program makes all the mode lines (except the last) scrollable vertically and then proceeds to scroll them. It does not check how far it's scrolled so far, so it will eventually start showing garbage on the screen. See SAVMSC at locations 88 and 89 for an explanation of why. Press SYSTEM RESET when you've had enough.

One thing you'll also see when you run this program is the main problem with fine scrolling from BASIC: You can't change the LMS address and VSCROL at exactly the same time, so the whole screen appears to "jump" down a line every so often. Although there is no way to get rid of this, you can use SDMCTL at location 559 to turn off the screen while you change the LMS. This generates a brief "flash" instead of the jump. See for yourself; add the following lines to the preceding program:

```
205 POKE 559,34
255 POKE 559,0
```

This is about the best you can do from BASIC.

In modes zero and one, where the characters are 8 scan lines high, VSCROL can vary from 0 to 7. In mode two, where the characters are 16 scan lines high, it can vary from 0 to 15.

Noname
54278 D406

This location is not used.

PMBASE (POKE only)
54279 D407

Back in GTIA/CTIA, we were discussing player/missile graphics. We discovered that we could either keep supplying GTIA/CTIA with the player/missile data ourselves, or have ANTIC do it for us. If ANTIC is doing it, them PMBASE is used to tell ANTIC where the data is stored. It is the high byte of the address, so the address itself is equal to the value you POKE into PMBASE times 256. Because of some esoteric requirements of ANTIC, PMBASE must be on a 2K boundary if you are using regular height players, and a 1K boundary if you are using double height players. How can you tell? If the value you are going to POKE into PMBASE is a multiple of four, then it's a 1K boundary. It has to be a multiple of eight to be a 2K boundary.

For a detailed explanation of how to use PMBASE and other player/missile graphics registers, see Appendix Two on – what else? – player/missile graphics.

Noname
54280 D408
Another location that isn’t used.

CHBASE (POKE only)
54281 D409

Another location with a shadow register that explains everything. See CHBAS at location 756 for a description of the character set address. Also see Appendix One on designing your own character sets.

WSYNC (POKE only)
54282 D40A

Storing any value in WSYNC will cause the 6502 to stop everything until the end of the current scan line (HBLANK). This is very useful if you want to synchronize something with the screen display. For an example, and more information, see VDSLST at locations 512 and 513. Note that VDSLST is not a shadow register for WSYNC.

VCOUNT (PEEK only)
54283 D40B

VCOUNT keeps track of what scan line is currently being drawn. Actually, it increases by one every two scan lines, so multiply the value by two to get the true number.

If you have more than one DLI, VCOUNT is a good way for your DLI routine to check which one is being processed. It can also be used to simulate DLIs. For example, you might write a loop that waits for VCOUNT to reach a certain value before going on. This allows you to spend more time in a certain routine than DLIs allow, but it also wastes a lot of time waiting.

There are a total of 262 scan lines on a screen (312 in Europe), so VCOUNT can range from 0 to 130 (155 in Europe).

PENH (PEEK only)
54284 D40C

This tells you the horizontal position of the light pen. See its shadow register, LPENH, at location 564 for more information.

PENV (PEEK only)
54285 D40D

Same as the preceding, except it's the vertical position and you should see LPENV at location 565.

NMIEN (POKE only)
54286 D40E

The last three bits (7-5) of NMIEN are used to enable or disable the NMIs. They are used as shown in Figure 48.

--1-----	(32)	enables the SYSTEM RESET interrupt.
-1------	(64)	enables the vertical blank interrupt.
1-------	(128)	enables the display list interrupt.

FIGURE 48. NMIEN bit chart

The OS initializes NMIEN to 64, thereby enabling vertical blank interrupts. It also sets NMIEN to 64 during the SETVBV routine mentioned at VVBLKD (548,549). So what? If you are writing a program where you will be using your own VBLANK routine and display list interrupts, make sure that you enable the display list interrupts after you use SETVBV. There have been a couple of times when I couldn't figure out why my DLIs weren't working, only to discover that I had enabled them before I set up my VBLANK routine.

A few of you out there may be thinking "What about SYSTEM RESET, isn't that an NMI as well?" Yes it is, but the computer does not allow you to disable it. Pressing SYSTEM RESET will always cause a warm start to occur. You can, however, store an address in DOSINI (12,13), since the OS jumps through DOSINI after it is done with the warm start. Most machine language programmers have DOSINI point to their program's initialization routine. That way, the program will start over again if someone presses SYSTEM RESET (normally the OS would go to BASIC or reboot the system).

NMIRES (POKE) and NMIST (PEEK)
54287 D40F

(POKE) POKEing any value here clears NMIST.

(PEEK) The last three bits of NMIST are used to identify what kind of interrupt has occurred (Figure 49).

--1-----	(32)	means that the SYSTEM RESET key has been pressed.
-1------	(64)	means that a vertical blank interrupt has occurred.
1-------	(128)	means that a display list interrupt has occurred.

FIGURE 49. NMIST bit chart

Unfortunately, since the OS has already take care of NMIST and WIRES by the time you can get to them, they don't really do you much good (you don't have to reset NMIST during your DLI routine).

Noname
54288-54783 D410-D5FF

These locations, the rest of ANTIC, are currently unused.

Noname
54784-55295 D600-D7FF

So are these.

THE OPERATING SYSTEM

Finally, way back at the end of memory, we come to the Operating System itself, stored in a 10K ROM cartridge (inside your computer). This OS ROM includes not only the program for the Operating System (yes, the OS is just another program), but also the floating point package, the data for the Atari character set, the device handlers, and various vectors.

As I’ve mentioned throughout the book, there are two versions of the OS as of this writing. Version “B” includes some changes to get rid of a few of the bugs that appeared in version “A.” These changes come mainly in SIO and the interrupt handler routines. The addresses I’ll be giving below will be for version “A,” since it is the best documented. You should see Appendix Five on OS changes to determine which locations will not be the same in version “B.” How do you know which version you have? If PEEK(58383) equals zero then you have version “B.”

If you need more specific information on the locations and routines described next, I suggest you study the appropriate parts of the OS Listing. It is well commented and relatively easy to understand.

All locations in the OS are PEEK only.

FLOATING POINT PACKAGE

Locations 55296 through 57343 hold the floating point package, a series of routines to do floating point math. For an explanation of how the Atari stores floating point numbers, see VVTP at locations 134 and 135. For information on the floating point registers, see locations 212 through 255. Finally, for information on the input and output buffers see INBUFF at 243 and 244, and LBUFF at 1408 through 1535.

The following is a list of some of the more useful routines in the package. The trigonometric functions are in the BASIC cartridge starting at location 48551.

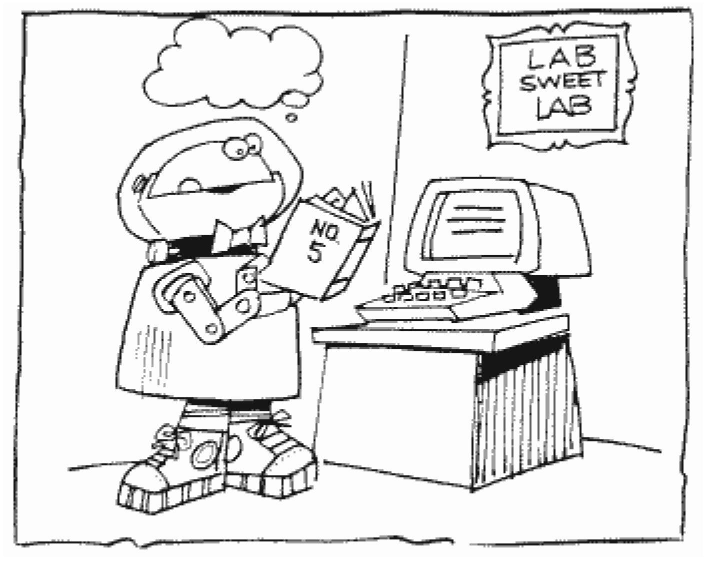
Note that in routines that use the carry bit (as indicated), if the carry is set at the end of the routine, then an error occurred. If it's clear, then everything is OK.

AFP
55296 D800

This routine takes an ATASCII representation of a number (i.e., "12345") and converts it to floating point (with carry). INBUFF points to the ATASCII number, floating point register zero (FR0) will hold the result.

You may be wondering why such a routine would be needed, and that's a very good thing to wonder. Suppose you had the following line in BASIC:

```
250 X =3.14159*37.5
```



When you type in such a line, BASIC sees the numbers as nothing more than a bunch of ATASCII characters. Before it can do the math, it must convert those characters into numbers it can understand. That's what AFP is for. BASIC will use AFP on both numbers (moving one of them to FR1), do the multiplication, and then store the result in

X. AFP is also needed for BASIC's STR$ function.

FASC
55526 D8E6

FASC does just the opposite of AFP. It takes a floating point number from FR0 and stores the ATASCII representation in LBUFF. This is necessary when a number needs to be printed on the screen, and also for BASIC's VAL function.

IFP
55722 D9AA

IFP is used to convert integers to floating point. It expects to see the integer in the first two bytes of FR0 (locations 212 and 213) and will store the result in FR0.

FPI
55762 D9D2

This does the exact opposite of IFP (with carry).

ZFR0
55876 DA44

ZFR0 sets all the bytes in FR0 to zero.

ZFI or AFI
55878 DA46

Sets FRx to zero, where x is the value in the X register.

FSUB
55904 DA60

FSUB subtracts FR1 from FR0 (with carry) and stores the result in FR0.

FADD
55910 DA66

Adds FR1 to FR0 (with carry) and stores the result in FR0 (notice that FADD is actually a part of FSUB).

FMUL
56027 DADB

Multiplies FR0 by FR1 (with carry) and stores the result in FR0.

FDIV
56104 DB28

Divides FR0 by FR1 (with carry) and stores the result in FR0.

PLYEVL
56640 DD40

This one is a little complicated, so bear with me. PLYEVL evaluates a polynomial, such as 5*Z^4+10*Z^2+2*Z+1 (read “five Z to the fourth plus ten Z squared plus two Z plus one”). For the sake of this routine, we’ll write such a polynomial as

SUM (I=N to 0) (A(I)*Z^I)

So in the preceding example, N=4, A(0) =1, A(1)=2, A(2) =10, A(3)=0 (since there is no Z cubed), and A(4)=5.

Why are we doing all of this? When you call PLYEVL, it expects you to provide the following information:

Somewhere in memory: a list of the A( ) values, in floating point format (BCD), starting with A(0).

X register: low byte of the starting address of the preceding list.

Y register: high byte of the starting address of the preceding list. Accumulator: N+1

FR0: Z

PLYEVL will take all of this and use it to evaluate the polynomial (with carry). The result will be stored in FR0.

FLD0R
56713 DD89

FLD0R will load FR0 with the floating point number pointed to by the X and Y registers. X should hold the low byte of the address of this number, Y the high.

FLD0P
56717 DD8D

FLD0P will load FR0 with the floating point number pointed to by FLPTR (252).

FLD1R
56728 DD98

FLD1R will load FR1 with the floating point number pointed to by the X and Y registers. X should hold the low byte of the address of this number, Y the high.

FLD1P
56732 DD9C

FLD1P will load FR1 with the floating point number pointed to by FLPTR (252).

FST0R
56743 DDA7

FST0R will store FR0 in memory, starting at the address pointed to by the X and Y registers. X should hold the low byte of this address, Y the high.

FST0P
56747 DDAB

FST0P will store FR0 in memory, starting at the address pointed to by FLPTR (252).

FMOVE
56758 DDB6

FMOVE moves the floating point number in FR0 to FR1.

EXP
56768 DDC0

EXP raises “e” to the FR0 power and stores the result in FR0 (FR0 = e^FR0).

EXP10
56780 DDCC

EXP10, as you may have guessed, raises 10 to the FR0 power and stores the result in FR0 (FR0=10^FR0). Notice that it is actually part of the EXP routine.

LOG
57037 DECD

LOG figures out the natural logarithm (base e) of FR0 and stores it back in FR0.

LOG10
57041 DED1

LOG 10 figures out the base 10 logarithm of FR0 and stores it back in FR0. Notice that it is part of the LOG routine.

THE CHARACTER SET

The data for the regular Atari character set is stored in locations 57344 through 58367 ($E000 to $E3FF). There are eight bytes for each character and 128 characters in all (for a grand total of 1024 bytes). But wait a minute. Doesn't the Atari have 256 characters? Yes, but the information for the regular characters is all the Atari needs to know to print inverse characters, so that's why there are only 128 character descriptions.

For lots of information on how the bytes are used to describe a character, and on the order of the characters within the character set, see CHBAS at location 756. For information on how to design your own character set, see the Appendix One, which is on that very topic.

The following program will use the character descriptions to put text on the screen in graphics mode eight:

```
100 GRAPHICS 8
105 SCRMEM=PEEK(88)+PEEK(89)*256
110 DIM TEXT$(120)
120 PRINT "Start text in what column (0-39)";
130 INPUT COL
140 PRINT "In what row (0-152)";
150 INPUT ROW
160 PRINT "Type in the text you want to print:"
170 INPUT TEXT$
180 CHSET=PEEK(756)*256
190 FOR CHAR=1 TO LEN(TEXT$)
200 ATASC=ASC(TEXT$(CHAR,CHAR))
210 NOINV=ATASC-128*(ATASC>127)
220 INTRNL=NOINV-32*(NOINV<96)+96*(NOINV<32)
230 FOR BYTE=CHSET+INTRNL*8 TO CHSET+INTRNL*8+7
240 POKE SCRMEM+ROW*40+COL,ABS(255*(ATASC>127)-PEEK(BYTE))
250 ROW=ROW+1
260 NEXT BYTE
270 ROW=ROW-8
280 COL=COL+1
290 IF COL=40 THEN COL=0:ROW=ROW+8
300 NEXT CHAR
310 PRINT
320 GOTO 120
```

VECTORS AND VECTOR TABLES

What are vector tables? You remember that a vector is a pair of memory locations that hold the address of a routine. A vector table is, quite simply, a table of vectors. Thus, locations 58368 through 58533 hold the addresses of various routines, mostly having to do with I/O or interrupts.

EDITRV
58368-58383   E400-E40F

This is the vector table for the screen editor handler. For a description of its contents, along with the contents of the next four vector tables, see HATABS at locations 794 through 831 (where we called it a "handler address table").

SCRENV
58384-58399   E410-E41F

The vector table for the display handler. See the note at EDITRV.

KEYBDV
58400-58415   E420-E42F

The vector table for the keyboard handler. See the note at EDITRV.

PRINTV
58416-58431   E430-E43F

The vector table for the printer handler. See the note at EDITRV.

CASETV
58432-58447   E440-E44F

The vector table for the cassette handler. See the note at EDITRV.

You will notice that the following 16 vectors are three bytes long rather than two. Why the extra byte? The first byte of each vector is a 6502 JMP instruction, while the address is in the second two bytes.

The purpose of these vectors may not be obvious to you (they weren't to me). Atari knew that it would probably need to make changes to the OS at some point. It also wanted to make sure that old programs would still be able to work with these newer versions of the OS, even though some of the addresses would be different. The solution was to use vectors. That way, even though the addresses in the vectors would change, the addresses of the vectors would remain the same, and programs using these addresses would still work. The reason that some programs don't work with version "B" of the OS is that these programs didn't use the vectors.

DISKIV
58448-58450 E450-E452

DISKIV is the initialization vector for the disk handler. It points to location 60906.

DISKINV
58451-58453 E453-E455

This is the entry vector for the disk handler. It points to location 60912.

CIOV
48454-48456 E456-E458

CIOV is the entry vector for CIO (Central Input/Output). See Appendix Seven on I/O for an explanation of what CIO does.

You can use CIO yourself by first setting up an IOCB (see locations 832 through 959), and then using the following routine:

```
100 DIM ML$(7)
110 GOSUB 10000
120 CIO=USR(ADR(ML$),IOCB*16)
130 END
10000 FOR BYT=1 TO 7
10010 READ INSTR
10020 ML$(BYT,BYT)=CHR$(INSTR)
10030 NEXT BYT
10040 RETURN
10050 DATA 104,104,104,170,32,86,228
```

The data is for this machine language routine:

```
68     PLA
68     PLA
68     PLA
AA     TAX
2056E4 JSR $E456
```

CIO expects the number of the IOCB you want to use, times 16, in the X register. That's why we have IOCB* 16 in the preceding program. You should substitute the IOCB number you are using for IOCB. Remember to OPEN the IOCB first.

CIOV points to 58564

SIOV
58457-58459   E459-E45B

This is the entry vector for SIO (Serial Input/Output). Again, check Appendix Seven on I/O for an explanation of SIO's function. SIOV points to 59737.

SETVBV
58460-58462   E45C-E45E

SETVBV is the entry vector for a routine that serves two purposes. First of all, as we saw at VVBLKD (548,549), it will set up VVBLKI and VVBLKD for us. Second, as we saw at CDTMA1 (550,551), it will also set up the vectors for the system timers. See VVBLKD and CDTMA1 for more information.

SETVBV points to 59666 in version "A" of the OS, 59629 in version "B."

SYSVBV
58463-58465   E45F-E461

This is the entry vector for the stage one VBLANK routine. Unless you have your own routine, VVBLKI (546,547) normally points here. See VVBLKI and VVBLKD (548,549) for more information on VBLANK.

SYSVBV points to 59345 in version "A" of the OS, 59310 in version "B."

XITVBV
58466-58468   E462-E464

XITVBV is the exit vector for the VBLANK routine. This is what VVBLKD points to unless you've changed it.

Use XITVBV to return to where the computer left off from when the VBLANK interrupt occurred. It points to 59710 in version "A" of the OS, 59653 in version "B."

The following four vectors are designed for use by the OS only.

SIOINV
58469-58471   E465-E467

This is the initialization vector for SIO.

SENDEV
58472-58474   E468-E46A

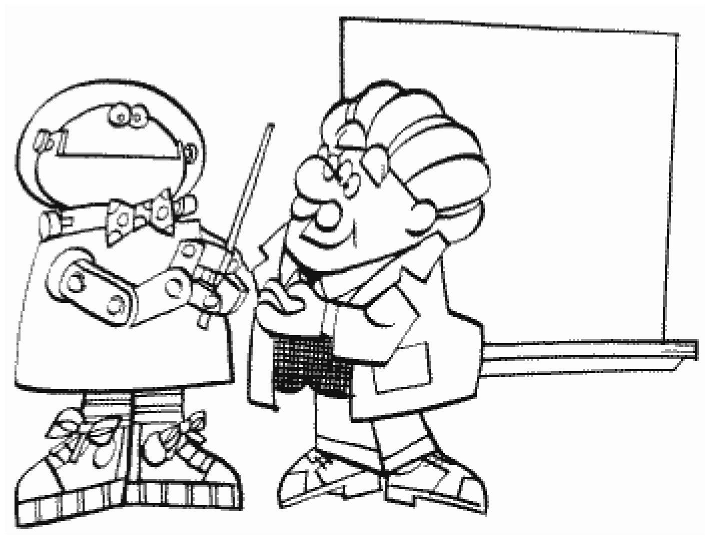
SENDEV is the vector for the "send-enable" routine.

INTINV
58475-58477   E46B-E46D

This is the initialization vector for the interrupt handler routine.

CIOINV
58478-58480   E46E-E470

CIOINV is the initialization vector for CIO.



BLKBDV
58481-58483   E471-E473

This is the entry vector for the blackboard mode, which is more commonly known as the "ATARI MEMO PAD" mode. Type "BYE" from BASIC, or turn on the computer with no cartridges or disk drives to see what I mean. This mode lets you type things on the screen without anybody caring what you type. In other words, you can press RETURN and nothing will happen. To get back to BASIC, press SYSTEM RESET (this won't erase your BASIC program).

BLKBDV points to 61987.

WARMSV
58484-58486   E474-E476

WARMSV is the entry vector for the warm start routine. The OS jumps through here when SYSTEM RESET is pressed.

WARMSV points to 61723.

In case these locations don't seen useful to you, try this:

X = USR(58484)

What you have just done is told the computer to go to 58484, which contains a machine language instruction to go to the address in the next two memory locations. Since this routine is for what's called warm start, the computer will now act just like you pressed SYSTEM RESET. You use the other locations in this section just like this. TRY IT!

COLDSV
58487-58489 E477-E479

This, appropriately, is the entry vector for the cold start routine. Whereas going through WARMSV only initializes the OS RAM, going through COLDSV initializes all RAM, meaning that any programs in memory will be erased. See COLDST at location 580 for a way to hook COLDSV up to SYSTEM RESET rather than WARMSV.

COLDSV points to 61733

The following two vectors are designed for use by the OS only.

RBLOKV
58490-58492 E47A-E47C

RBLOKV is the entry vector for the cassette "read-block" routine.

CSOPIV
58493-58495 E47D-E47F

This is the vector for the cassette "OPEN-for-input" routine.

VCTABL
58496-58533 E480-E4A5

VCTABL is a table of the initial values for the OS RAM vectors.

Now we're into the final part of the OS, which consists mostly of the various built-in handlers, interrupt routines, and so forth. What follows is a list of addresses for some of these routines, which can be useful to you in one of several ways. If you're a beginner, the list will provide you with an idea of exactly what the OS does. If you're a machine language programmer then, along with the OS listing, the list will help you find the various routines so that you can see exactly how things are done. By studying the routines, you can also pick up on programming techniques (don't be afraid of the OS listing; it's really not that difficult to understand). Finally, if you really know what you're doing, you can rewrite the routines and put them in your own programs, customizing them to your own needs.

Most of the routines will **not** work without some kind of previous setup, so make sure you check the OS listing before you attempt to use them.

Please note that all the following addresses are for the original OS only. Some of them may be different in the newer versions. At the time of this writing, however, the OS listing is for the original version, and that is why these addresses are used.

CIO ROUTINES

CIOINT
58534 E4A6

CIO's initialization routine.

CIO
58564 E4C4

The main CIO routine (includes the following routines).

CIOPEN
58633 E509

OPEN routine.

CICLOS
58675 E533

CLOSE routine.

CISTSP
58702 E54E

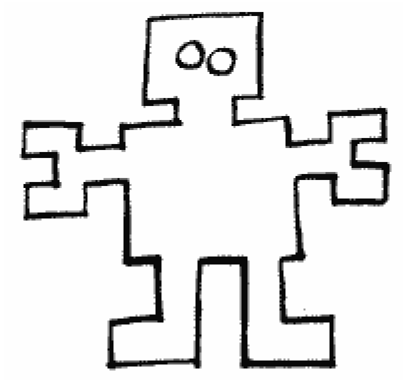
STATUS and special requests routine.

CIREAD
58729 E569

GET routine (GET character and GET record).

CIWRIT
58825 E5C9

PUT routine (PUT character and PUT record).



CIRTN1
58907 E61B

Return from CIO with the status in the Y register.

CIRTN2
58909 E61D

Return from CIO with the status in ICSTAZ (35).

COMENT
58941 E63D

Compute the handler entry point using HATABS (794) and COMTAB (59081).

GOHAND
59017 E689

Jump indirectly to the device handler. An indirect jump, in this case, means fooling the 6502 into thinking that the address you want to jump to is actually the one you want to RTS to. This involves playing with the stack and is a pretty neat trick you may want to look at.

DEVSRC
59038 E69E

Find a particular device in the handler address table.

COMTAB
59081 E6C9

This is a table of offsets into the handler entry point table for the desired device. It is used to find the correct vector for the given command.

INTERRUPT HANDLER ROUTINES

IHINIT
59093 E6D5

Initialize the interrupt handlers.

PIRQ
59123 E6F3

Jump to the main IRQ handler routine through VIMIRQ (534,535). Unless you've changed it, VIMIRQ points to SYIRQ.

SYIRQ
59126 E6F6

This is the system's IRQ handler routine.

PNMI
59316 E7B4

This is the system's NMI handler routine.

SYSTEM VBLANK ROUTINES

SYSVBL
59345 E7D1

This is the immediate vertical blank routine (stage one VBLANK).

SYSVB3
59400 E808

This is the stage two VBLANK routine.

SETVBL
59666 E912

This routine can be used to set up vectors for your own VBLANK routines, and also for the system timers. See SETVBV at 58460.

XITVBL
59710 E93E

Exit from vertical blank.

SIO ROUTINES

SIOINT
59716 E944

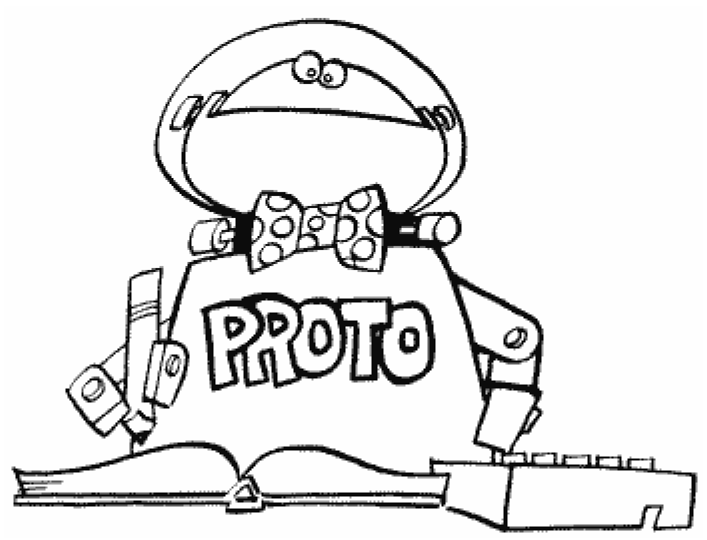
SIO's initialization routine.

SIO
59737 E959

The main SIO routine (includes the following routines).

RETURN
59917 EA0D

Return from SIO.



WAIT
59930 EA1A

Wait for the device to finish what it has been told to do.

SEND
60011 EA6B

Send a buffer of bytes to a device.

ISRODN
60048 EA90

This is the “serial output data needed” interrupt routine. See SEROUT at location 53773.

ISRTD
60113 EAD1

This is the “transmission done interrupt” routine. See POKMSK at location 16.

RECEIV
60130 EAE2

Receive a bunch of bytes from a device and store them in a buffer.

ISRSIR
60177 EB11

This is the “serial input data needed” interrupt routine. See SERIN at location 53773.

CASENT
60292 EB84

Read or write a record to cassette (SIO handles the cassette differently than other devices).

BEGIN
60692 ED14

Figure out the baud rate for the next record. See CBAUDL/H at locations 750 and 751.

POKTAB
60882 EDD2

This is a table of values used in the preceding baud rate routine.

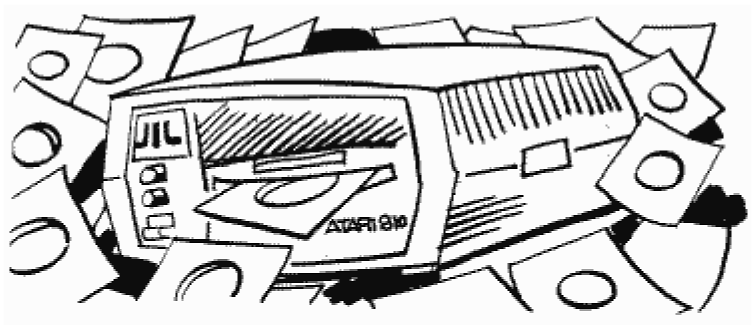
DISK INTERFACE ROUTINES

DINIT
60906 EDEA

The disk interface's initialization routine.

DSKIF
60912 EDF0

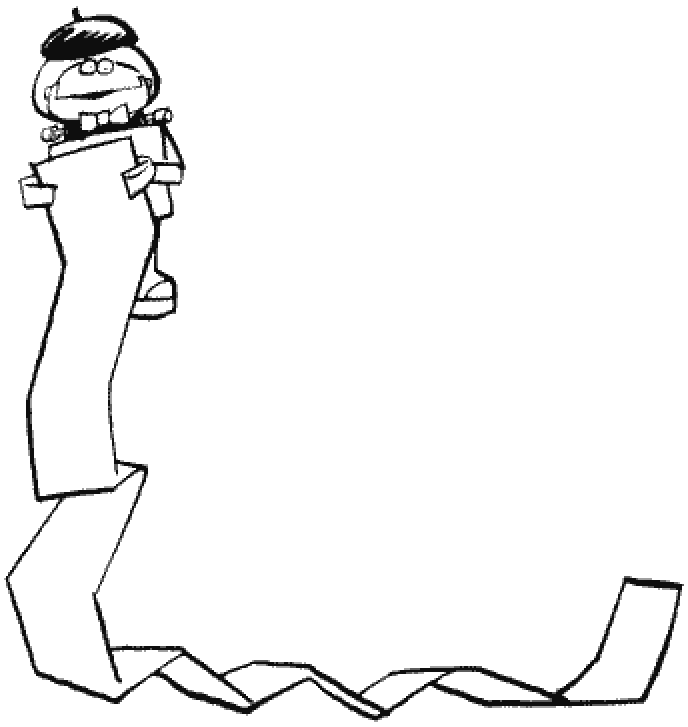
The main disk interface routine.



PRINTER HANDLER ROUTINES

PRNORG
61048 EE78

This is the beginning of the printer handler. See HATABS at location 794 for a list of routines in this and any handler.



CASSETTE HANDLER ROUTINES

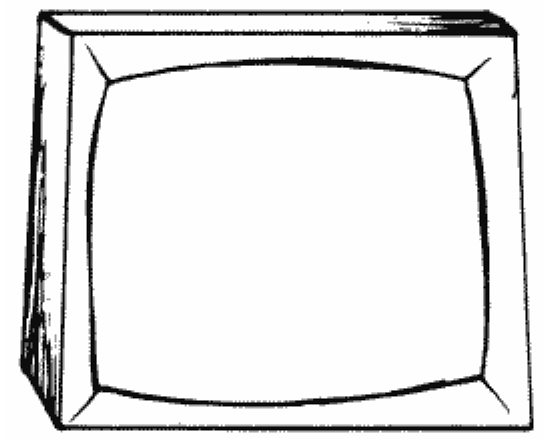
CASORG
61249 EF41

This is the beginning of the cassette handler. See the note at PRNORG.

BEEP
61528 F058

The cassette handler uses this routine to make the keyboard speaker “beep” when you type CLOAD or CSAVE.

MONITOR ROUTINES



RESET
61723 F11B

This is the start of the SYSTEM RESET routine.

PWRUP
61733 F125

The start of the cold start routine.

ZERORM
61752 F138

Clear all the RAM locations.

ZOSRAM
61792 F160

Clear the OS RAM only (for warm start).

BLACKB
61994 F22A

The blackboard routine (MEMO PAD mode).

SPECL
62015 F23F

Check to see how much RAM there is.

HARDI
62081 F281

Initialize the hardware locations.

OSRAM
62100 F294

Initialize the OS RAM locations.

BOOT
62159 F2CF

Boot the disk if it's so desired (i.e., the disk drive is hooked up and turned on).

CSBOOT
62386 F3B2

Boot the cassette if it's so desired (i.e., the START button was held down when the computer was turned on).

DISPLAY HANDLER ROUTINES

DOPEN
62454 F3F6

OPEN the display handler (set up a graphics mode).

GETCH
62867 F593

GET a character from the screen.

OUTCH
62903 F5B7

PUT a character on the screen.

OUTPLT
62944 F5E0

PLOT a point on the screen.

SCREEN EDITOR ROUTINES

EGETCH
63038 F63E

INPUT a logical line from the keyboard and print it to the screen. Remember that a logical line ends either when you press return or fill three rows on the screen.

EOUTCH
63140 F6A4

PRINT a character on the screen, making sure that control characters are processed instead of just printed (i.e., a CTRL-arrow will move the cursor rather than printing an arrow).

KEYBOARD HANDLER ROUTINES

KGETC2
63197 F6DD

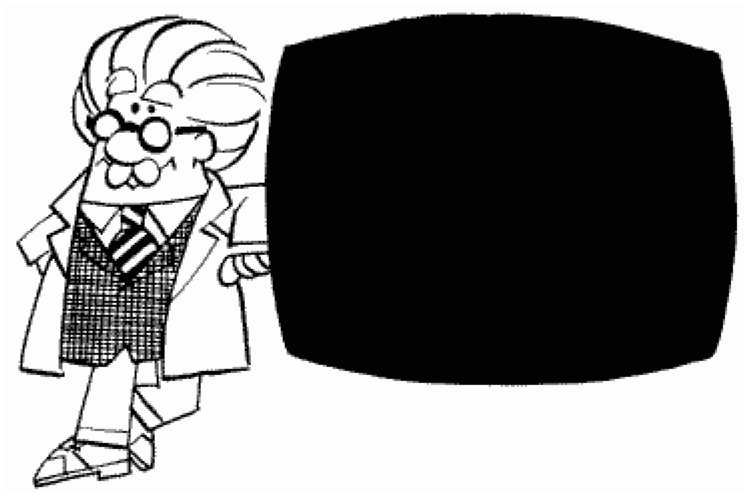
GET a character from the keyboard.

ESCAPE
63353 F779

Process all the various control characters.

BELL
63754 F90A

Ring the bell.



MORE DISPLAY HANDLER ROUTINES

CONVRT
63815 F947

Takes the row number and column number that the cursor is on and figure out what memory location that corresponds to.

INATAC
64306 FB32

Convert an internal character value to its ATASCII value.

CLRLIN
64411 FB9B

Clear the line that the cursor is currently on.

SCROLL
64428 FBAC

Scroll the screen.

DRAW
64764 FCFC

Draw a line from OLDROW,OLDCOL to ROWCRS,COLCRS (locations 90 through 92 and 84 through 86).

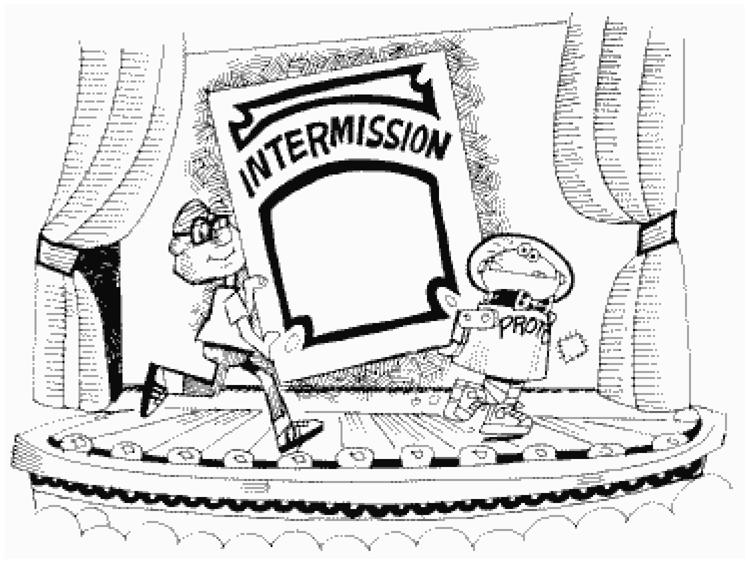
TABLES, TABLES, AND MORE TABLES

Locations 65093 through 65469 are various tables for use with the display handler. Check the OS listing for more details and to find out which routines use them (use the cross-reference table at the end of the listing).

ONE MORE KEYBOARD ROUTINE

PIRQQ
65470 FFBE

The "IRQ" in this location's name should tip you off to the fact that this is the interrupt routine for the keyboard. It debounces the keys, checks for CTRL-1 (pause) and sets SSFLAG accordingly (767), stores the key value in CH (764) and CH1 (754), and clears ATRACT (77).



THAT’S ALL FOLKS

Yup, that brings us to the end of Atari memory. Thanks for bearing with me for all of this. You can now relax and take a well-deserved break before going on to the appendices.

APPENDIX ONE

DESIGNING YOUR OWN CHARACTER SETS

Make sure you have read the description of CHBAS at location 756, RAMTOP at location 106, and the section on the character set at location 57344 before you attempt to use this appendix.

As you can imagine, being able to redefine the character set opens up a lot of graphics capabilities. You can create special graphics characters, do simple animation, or just make the letters look nicer. It's not even that difficult to do, so let's get right into it.

The first step is to decide what you want your new characters to look like. This is the step that's going to take the most time, since there is a total of 128 characters. Of course, you don't have to change them all, but if you do you will have to decide which of over 8,000 dots you want on, and which you want off. To help you out in this task, even if you're only changing a few, you may want to purchase a character editor. The one put out by Educational Software, Inc. is probably one of the best and even comes with a software tutorial on how to use custom character sets. In any case, for the sake of this explanation we'll only change four characters (we'll also see how to use the Atari characters for the ones we don't change). Figure 50 is how we are going to want our first three characters to look.

```
      ####                ####          ####
   #####                  ##            ##
  ######               #####            ##
  ######              ######         #####
   ####               ######        ######
                       ####         ######
                                     ####
```

FIGURE 50. Three notes

These funny-looking things are going to create a bouncing musical note when we're done with them. Before we continue, however, you should notice that the largest one only uses seven bytes. Most Atari characters, in fact, only use six, and are only six bits wide. Why? When we print characters side by side, especially letters, most of the time we don't want them to touch. By leaving the first and last bytes and first and last bits blank, we make sure that they won't. If we have a situation where we need them to touch, then all the bits and bytes can be used. Another thing to notice is that everything is at least two bits wide. This is because of artifacting, which we ran into at COLOR1 (709). Since each pixel in a graphics mode zero character is the same size as a pixel in graphics mode eight, we run into artifacting problems here also. If lines are not at least two pixels wide, then the line will not be white. We'll see this with our fourth character (Figure 51).

```
               # #
               #
               #
             # #
           # # #
           # # #
             #
```

FIGURE 51. Single note

OK, so now we know what we want our characters to look like. The next step is to convert them into numbers the computer can understand. This is another benefit of the character editor, since it will do the conversion automatically. If you're doing it by hand, just remember that each pixel represents a bit. Also, don't forget to include bytes with a value of zero. You have to end up with eight bytes for each character. With all this in mind, let's convert our four characters (Figure 52).

####	=	00001111	=	15
#####	=	01111100	=	124
######	=	11111100	=	252
######	=	11111100	=	252
####	=	01111000	=	120
	=	00000000	=	0
	=	00000000	=	0
	=	00000000	=	0
####	=	00001111	=	15
##	=	00001100	=	12
#####	=	01111100	=	124
######	=	11111100	=	252
######	=	11111100	=	252
####	=	01111000	=	120
	=	00000000	=	0
	=	00000000	=	0
####	=	00001111	=	15
##	=	00001100	=	12
##	=	00001100	=	12
#####	=	01111100	=	124
######	=	11111100	=	252
######	=	11111100	=	252
####	=	01111000	=	120
	=	00000000	=	0
# #	=	00001010	=	010
#	=	00001000	=	8
#	=	00001000	=	8
# #	=	00101000	=	40
# # #	=	10101000	=	168
# # #	=	10101000	=	168
#	=	00100000	=	32
	=	00000000	=	0

FIGURE 52. Note character bit chart

Now we're all set to put it somewhere. But where? The regular character set is in ROM, so we can't put it there. We have to protect a segment of RAM that we can use for the character set. The easiest way to do this is to use RAMTOP at location 106 and RAMSIZ at location 740. This is done by the following program lines:

```
100 SETNEW=PEEK(740)-4
110 POKE 106,SETNEW-4
120 GRAPHICS 0
```

The reason we use RAMSIZ is because if we didn't, and had NEWSET=PEEK(106)-4 instead, we would change RAMTOP every time we ran the program (that's because RAMTOP doesn't get set back to its original value unless you press SYSTEM RESET). So if we ran the program four times, RAMTOP would be decreased by 32 from its original value. If we ran it enough times, we would eventually get an out-of-memory error (try it), so we use RAMSIZ as well. Anyway, this gives us 2K of protected RAM to work with. Remember that under some circumstances the first 1K isn't really safe, so we'll be using the second 1K for our character set.

Since we're only changing four characters, we'll want to use some of the regular character set, so let's move it from ROM to RAM:

```
130 CHSET=57344
140 FOR BYTE=0 TO 1023
150 POKE SETNEW*256+BYTE,PEEK(CHSET+BYTE)
160 NEXT BYTE
```

Wow, it takes a long time to move 1024 bytes, doesn't it? Let's do that again in machine language:

```
10 DIM MM$(41)
20 FOR CHAR=1 TO 41
30 READ CODE
40 MM$(CHAR,CHAR)=CHR$(CODE)
50 NEXT CHAR
60 DATA 104,104,133,204,104,133,203,104,133,206,104
70 DATA 133,205,104,170,160,255,138,208,2,104,168,177
80 DATA 203,145,205,136,192,255,208,247,230,204,230
90 DATA 206,202,224,255,208,231,96
130 CHSET=57344
140 X=USR(ADR(MM$),CHSET,SETNEW*256,1024)
150 REM
160 REM
```

Before I give you the assembly language code for the machine language routine stored in MM$, let me tell you a little bit about it. MM stands for "Move Memory," and I wrote the routine so you can use it for other things as well. Your USR statement should have the following format for MM$:

X=USR(ADR(MM$), *FROM*, *TO*, *NUMBER*)

FROM is the starting address you want to move from, TO is the starting address you want to move to, and NUMBER is the number of bytes you want to move. Note that MM$ will not work properly if FROM+NUMBER is less than TO, or if TO+256 is less than FROM.

Here's what the routine looks like:

```
68            PLA
68            PLA
85CC          STA FROMHI
68            PLA
85CB          STA FROMLO
68            PLA
85CE          STA TOHI
68            PLA
85CD          STA TOLO
68            PLA
AA            TAX
A0FF LOOP 1   LDY #255
8A            TXA
D002          BNE LOOP2
68            PLA
A8            TAY
B1CB LOOP 2   LDA (FROMLO),Y
91CD          STA (TOLO),Y
88            DEY
C0FF          CPY #255
D0F7          BNE LOOP2
E6CC          INC FROMHI
E6CE          INC TOHI
CA            DEX
E0FF          CPX #255
D0E7          BNE LOOP1
60            RTS
```

There, that's much better. Now the Atari character set is in the RAM area that we protected. You can verify this by changing CHBAS:

```
 POKE 756, SETNEW
```

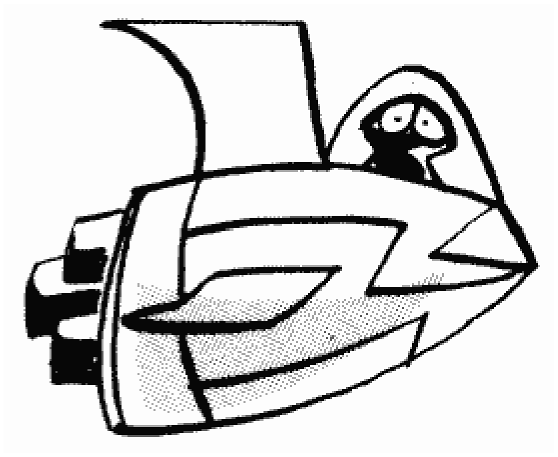
The next step is to decide which of the characters we want to change to our new characters. There's no point in messing up any of the letters here, so let's change CTRL-A through CTRL-D. These are normally Atari graphics characters and have ATASCII values of 1 through 4. You'll recall, however, that the character set has a different order than ATASCII. Looking at Appendix Nine, we see that the characters we want have internal values of 65 through 68. We're all set now to make the changes:

```
170 SET=SETNEW*256
180 FOR CHAR=65 TO 68
190 FOR BYTE=0 TO 7
200 READ DAT
210 POKE SET+CHAR*8+BYTE,DAT
220 NEXT BYTE
230 NEXT CHAR

1000 DATA 15,124,252,252,120,0,0,0
1010 DATA 15,12,124,252,252,120,0,0
1020 DATA 15,12,12,124,252,252,120,0
1030 DATA 10,8,8,40,168,168,32,0
```

Our character set is now all ready to be used, so let's tell the computer where it is:

```
240 POKE 756,SETNEW
```



And let's use it:

```
250 POKE 752,1
260 POSITION 10,4
270 PRINT CHR$(4)
280 RESTORE 1100
290 FOR LP=1 TO 4
300 READ CHAR
310 POSITION 10,2
320 PRINT CHR$(CHAR);
330 FOR WAIT=1 TO 25
340 NEXT WAIT
350 NEXT LP
360 GOTO 280
```

```
1100 DATA 1,2,3,1
```

Notice that all we have to do to animate the note is print the different versions one after another in the same place, with a small delay between them to slow things down. This technique is used by a lot of programs to do simple (and sometimes complex) animation. For example, the aliens in Atari's Space Invaders program are created using a redefined character set. So don't feel you just have to use character sets for letters. By the way, the artifacted note is printed by itself so you can see what it looks like. It isn't supposed to be doing anything.

Make the following changes to the preceding program if you want to try the character set in graphics modes one or two:

```
120 GRAPHICS 1
240 POKE 756,SETNEW+2
270 PRINT #6;CHR$(4)
320 PRINT #6;CHR$(CHAR)
```

You can get rid of the hearts by redefining the heart character to a space. To find out why they're there and why we need NEWSET+2, see CHBAS. Incidentally, I said at CHBAS in the same section that you can't have upper and lowercase letters at the same time in graphics modes one and two. I lied. All you have to do is redefine the character set so that the graphics characters become uppercase letters. You can even use MM$ to do this. The following statement will move the uppercase letters into the graphics characters:

X=USR(ADR(MM$),57377, SETNEW*256+65,26)

Now a CTRL-A will give you an uppercase "A".

A few final bits of information. If you're using graphics mode seven or eight, you may have to move RAMTOP by 16 pages instead of 8. This is because of a limitation of the system. I personally have never had any problems, but other people have. Just be aware of it. You should also be aware that a GRAPHICS command will restore CHBAS to its original value. Make sure you reset it after each GRAPHICS command if you want to keep using your character set. Also, SYSTEM RESET will reset CHBAS and RAMTOP, thereby destroying the character set completely. So stay away from SYSTEM RESET.

PROTO

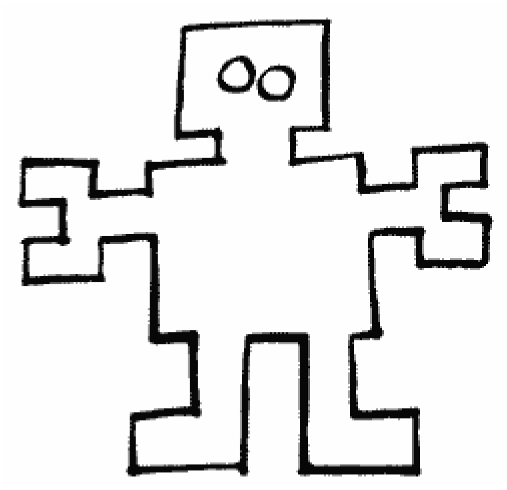
APPENDIX TWO

PLAYER/MISSILE GRAPHICS

When I first started programming microcomputers there was no such thing as player/missile graphics. You had to do animation using PLOTS and DRAWTOs, or by pretending that ">!<" was the Starship Enterprise. Even when I first started programming the Atari, some four years ago, player/missile graphics were just a rumor, even through the Atari was capable of doing them. That's right, Atari didn't tell anyone how to use player/missile graphics until months after the Atari was released, and even then you still had to do a lot of guessing. Anyway, so much for the old days. This appendix should contain all the information you need to be able to have all sorts of things flying around the screen with the greatest of ease.

I'll assume here that you've already read up on the various player/missile locations in the GTIA/CTIA chip, starting at location 53248, and on DMACTL and PMBASE in the ANTIC chip at 54272 and 54279. If you haven't then do so now.

Let's start by looking at a player.



WHAT IS A PLAYER?

What is a player? Well, that is easy to answer if you are talking baseball, but a little harder to explain for a computer. We draw shapes on the screen by turning on and off bits in memory. The area of memory with these bits is called screen memory. Now the bad part of using this method to draw pictures on the screen comes when you want to animate little shapes ON TOP OF THE SCREEN PICTURE. Say you want to draw a little man and make him walk across a screen that you have carefully drawn a landscape on. As he walks, his pixels have to replace the ones of the landscape ONLY WHERE THE TWO SHAPES OVERLAP. Then, when he moves a little further across the screen, you have to replace the landscape because you want the man shape somewhere else. WOW. That is a lot of pixels to turn on and off.

Players provide a better method. You simply store a shape (the little man) somewhere other than in screen memory and tell the Atari's special chips to put the shape on top or underneath the screen data. It will figure out things like overlapping pixels automatically. All you have to do is tell the computer the options you want. Here is an example.

A normal player is as tall as the screen and as wide as a graphics mode one character (eight bits = one memory location). It can be either 256 bytes high, in which case each pixel in it is the height of a graphics mode eight row (one scan line), or it can be 128 bytes high, in which case each pixel is as high as a graphics mode seven row (two scan lines). Since the shape of a player is described in the same fashion as that of a character, you may wish to think of a player as an abnormally high character. There are four players altogether and a fifth player that can be divided into four missiles (Figure 53).

A missile is the exact same thing as a player, except it is only two bits wide. Since there are four missiles, and eight bits in a byte, all four missiles are stored in the same byte. Each byte is used as in Figure 54.

So, for example, if you wanted both pixels of missile one to be on in a particular byte, and weren't using any of the other missiles, you would set that byte to 00001100, which corresponds to a value of 12. You can also turn on only one bit of a missile to have thin missiles.

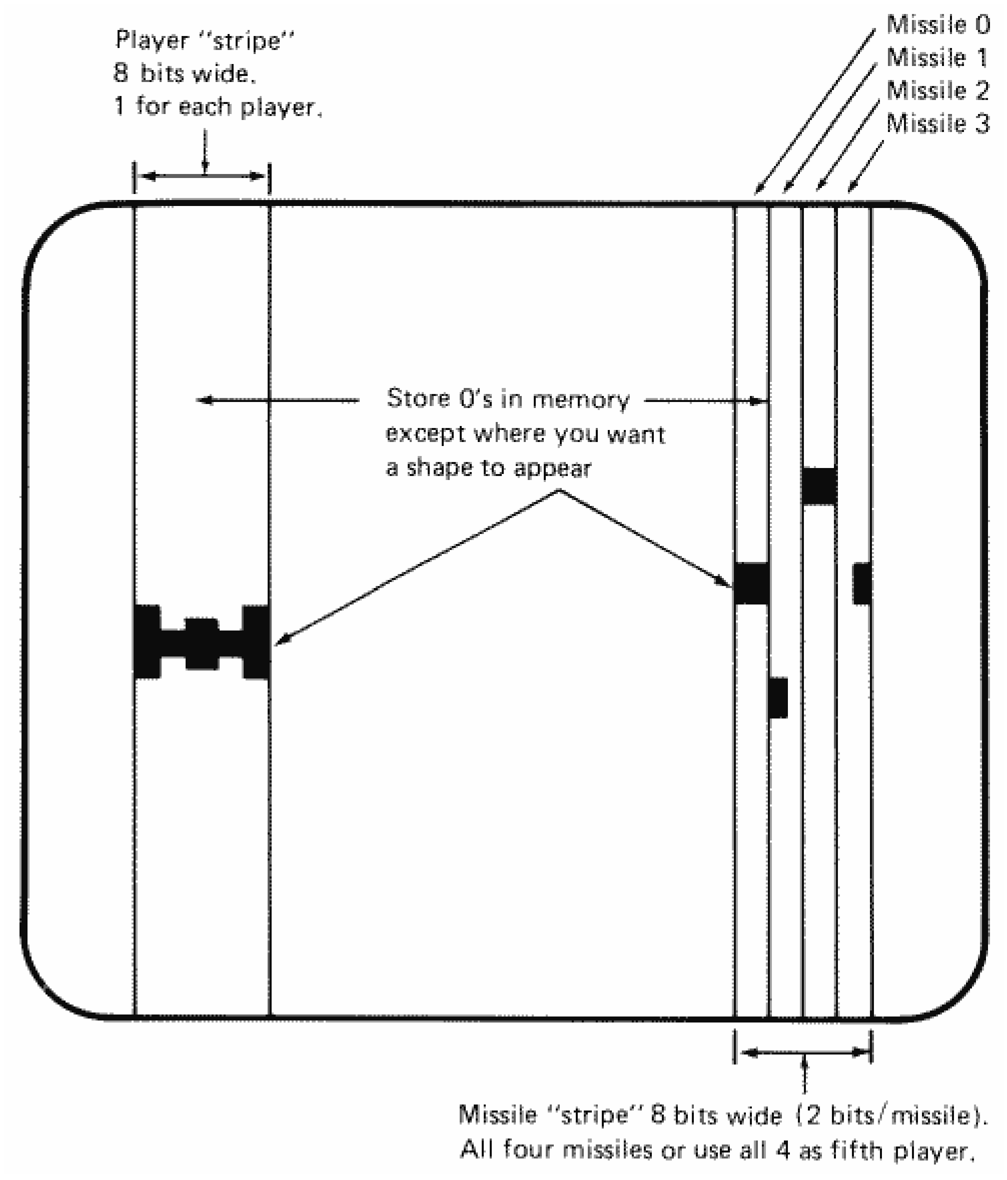
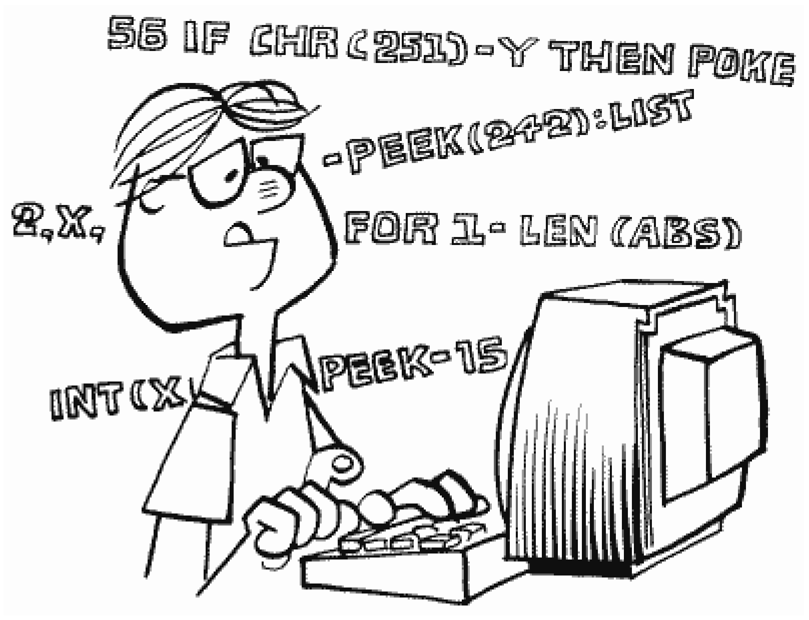

Player "stripe"
8 bits wide.
1 for each player.
Missile 0
Missile 1
Missile 2
Missile 3
Store 0's in memory except where you want a shape to appear
Missile "stripe" 8 bits wide (2 bits/missile).
All four missiles or use all 4 as fifth player.

FIGURE 53. Player missile positioning

BIT	7 6	5 4	3 2	1 0
USE	M3	M2	M1	M0

FIGURE 54. Storing missiles



PLAYER RESOLUTION AND WHERE TO STICK THEM

So we have a total of four players, each 1 byte wide and either 128 or 256 high, and four missiles, each 2 bits wide and either 128 or 256 bytes high. That means we'll need either 5*128 = 640 or 5*256 = 1280 bytes to store their descriptions. Before we look at where to find these bytes, let's look at how to choose whether we want the players and missiles to be 128 or 256 bytes high. The location that'll do the trick here is called SDMCTL (559). Setting bit four of SDMCTL chooses 256 byte player/missiles, while leaving it clear chooses 128 byte ones. So much for that, let's get back to our memory problem. Before I tell you where to get the memory from, there are a few more things you need to know. First of all, ANTIC relies on PMBASE at location 54279 to tell it where the memory is. Now ANTIC and PMBASE have some quirks that limit where the memory can be. First of all, PMBASE gives the high byte of the address only, which means that it will represent a multiple of 256. Furthermore, ANTIC says that it has to be on a 1K boundary for 128 byte player/missiles (PMBASE is a multiple of four), and on a 2K boundary for 256 byte player/missiles (PMBASE is a multiple of eight). And if that's not enough, ANTIC insists on there being extra bytes between PMBASE and the beginning of the player/missile data. Here's how ANTIC expects the data to be laid out (Figure 55a and 55b).

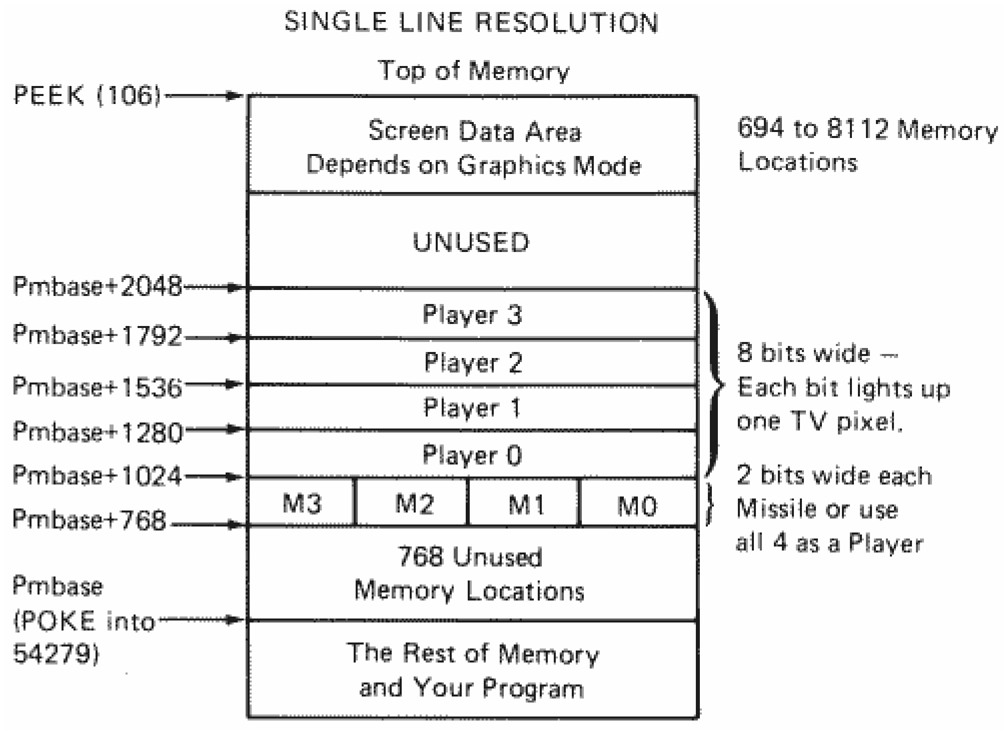

SINGLE LINE RESOLUTION
Top of Memory
PEEK (106)
Screen Data Area Depends on Graphics Mode
694 to 8112 Memory Locations
UNUSED
Pmbase+2048
Player 3
Pmbase+1792
Player 2
Pmbase+1536
Player 1
Pmbase+1280
Player 0
Pmbase+1024
M3
M2
M1
M0
Pmbase+768
8 bits wide – Each bit lights up one TV pixel.
2 bits wide each Missile or use all 4 as a Player
768 Unused Memory Locations
Pmbase (POKE into 54279)
The Rest of Memory and Your Program

FIGURE 55a. Single-line resolution

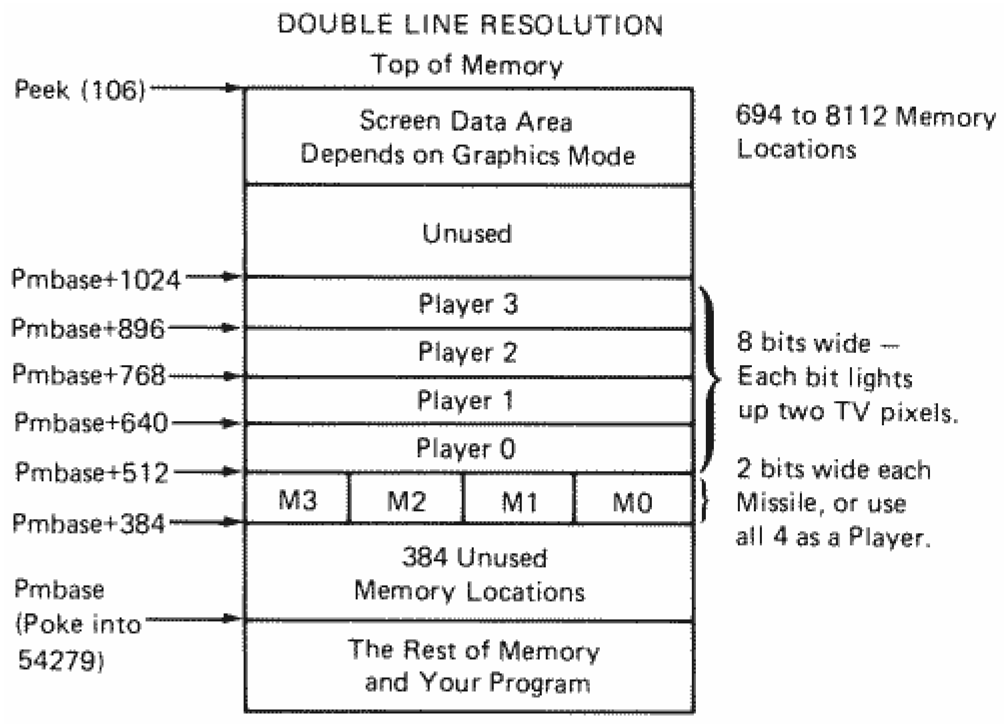

DOUBLE LINE RESOLUTION
Top of Memory
Peek (106)
Screen Data Area Depends on Graphics Mode
694 to 8112 Memory Locations
Unused
Pmbase+1024
Player 3
Pmbase+896
Player 2
Pmbase+768
Player 1
Pmbase+640
Player 0
Pmbase+512
M3
M2
M1
M0
Pmbase+384
8 bits wide – Each bit lights up two TV pixels.
2 bits wide each Missile, or use all 4 as a Player.
384 Unused Memory Locations
Pmbase (Poke into 54279)
The Rest of Memory and Your Program

FIGURE 55b. Double-line resolution

Now I’m finally going to tell you where to get the free memory from. You might even have guessed already since we encountered the same need in Appendix One. That’s right, we’ll change RAMTOP at location 106. I realize that I went through a lot of information just to tell you that, but I wanted to make sure you knew what we were going to have to do to RAMTOP, and why. At least we can start our example now. We’ll use 256 byte player/missiles, which means that we want to reserve eight pages using RAMTOP. You’ll recall from the description of RAMTOP that we have to reserve an extra four pages for safety’s sake, so we’ll change RAMTOP by 12 altogether:

```
100 PMBAS=PEEK(740)-8
110 POKE 106,PMBAS-4
120 GRAPHICS 19
```

There are two things to watch out for here. If you’re using graphics mode seven or eight, you should move RAMTOP by 16. The reason for this is given at RAMTOP. If you’re using a redefined character set as well as player/missile graphics, you should do the following:

```
PMBAS=PEEK(740)-8
CHBAS=PMBAS-4
POKE 106,CHBAS-4
GRAPHICS whatever
```

This will make sure that PMBASE sits on a 2K boundary, and at the same time will take care of the graphics seven and eight problem.

Before we tell ANTIC where we’ve decided to put the data, we should make sure that any old data is removed. If you don’t, you will have pixels on the screen you didn’t expect to see:

```
130 PM=PMBAS*256
140 FOR BYTE=PM+768 TO PM+2047
150 POKE BYTE,0
160 NEXT BYTE
```

Too slow for you? Try this:

```
10 DIM CM$(36)
20 FOR CHAR=1 TO 36
30 READ CODE
40 CM$(CHAR,CHAR)=CHR$(CODE)
50 NEXT CHAR
60 DATA 104,104,133,204,104,133,203,104,170,169,0,160
70 DATA 255,224,0,208,4,104,168,169,0,145,203,136,192
80 DATA 255,208,249,230,204,202,224,255,208,234,96
140 X=USR(ADR(CM$),PM+768,1280)
150 REM
```

```
160 REM
```

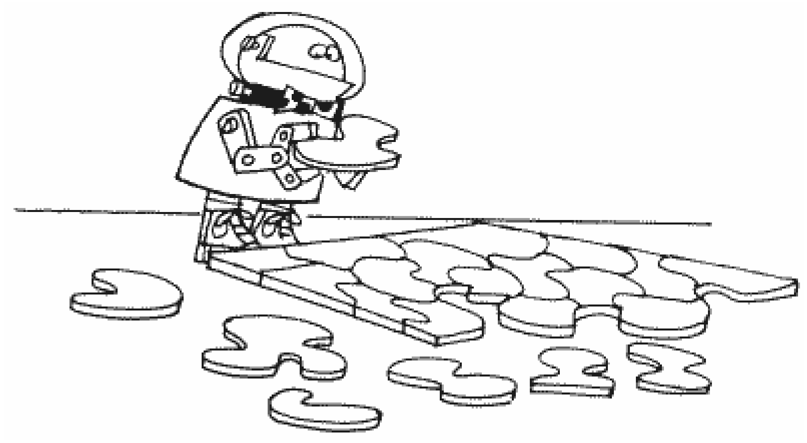
If you want to use CM$ for other things, it needs the following USR call:

X = USR(ADR(CM$), *START*,*NUMBER*)

START is the starting address of the memory you want to clear and NUMBER is the number of bytes to clear.

For the experts in the crowd, here's the assembly code for the machine language routine in CM$:

```
68            PLA
68            PLA
85CC          STA STARTHI
68            PLA
85CB          STA STARTLO
68            PLA
AA            TAX
A900          LDA #0
A0FF          LDY #255
E000  LOOP1   CPX #0
D004          BNE LOOP2
68            PLA
A8            TAY
A900          LDA #0
91CB  LOOP2   STA (STARTLO),X
88            DEY
C0FF          CPY #255
D0F9          BNE LOOP2
E6CC          INC STARTHI
CA            DEX
E0FF          CPX #255
D0EA          BNE LOOP1
60            RTS
```



Let’s complete the example we have started. We now tell ANTIC where the data will be:

```
170 POKE 54279,PMBAS
```

We also want to tell it to send the data to GTIA/CTIA, and that we will be using 256 byte player/missiles. This is done with SDMCTL:

```
180 POKE 559,62
```

We should also let GTIA/CTIA know that the data will be coming. GRACTL does this:

```
190 POKE 53277,3
```

All of this essentially serves to initialize player/missile graphics. Now we’re ready to go ahead and use them. Let’s start by choosing the shape of a player. We’ll make it look like a spaceship (Figure 56a).

Notice that we don’t have to use all 256 bytes. As a matter of fact, hardly anybody ever does, and we’ll see why in a little bit. First, let’s translate our picture to bytes (Figure 56b).

```
#   ##   #
# #### #
########
# #### #
#   ##   #
```

FIGURE 56a. Choosing your PM shape

```
#  ##  # = 10011001 = 153
# #### # = 10111101 = 189
######## = 11111111 = 255
# #### # = 10111101 = 189
#  ##  # = 10011001 = 153
```

FIGURE 56b. Mapping your PM

We'll store these bytes in the middle of player zero:

```
200 FOR BYTE=126 TO 130
210 READ DAT
220 POKE PM+1024+BYTE,DAT
230 NEXT BYTE

1000 DATA 153,189,255,189,153
```

Can you see the player on the screen now? I can't either. Why not? We forgot to tell GTIA/CTIA where we want it to appear on the screen. HPOSP0 at location 53248 will take care of that:

```
240 POKE 53248,128
```

Now it should be in the middle of the screen. Where is it? It is there, actually, but it has the same color as the background, so we can't see it. Let's take care of that by giving it a different color using PCOLR0 at location 704:

```
250 POKE 704,8
```

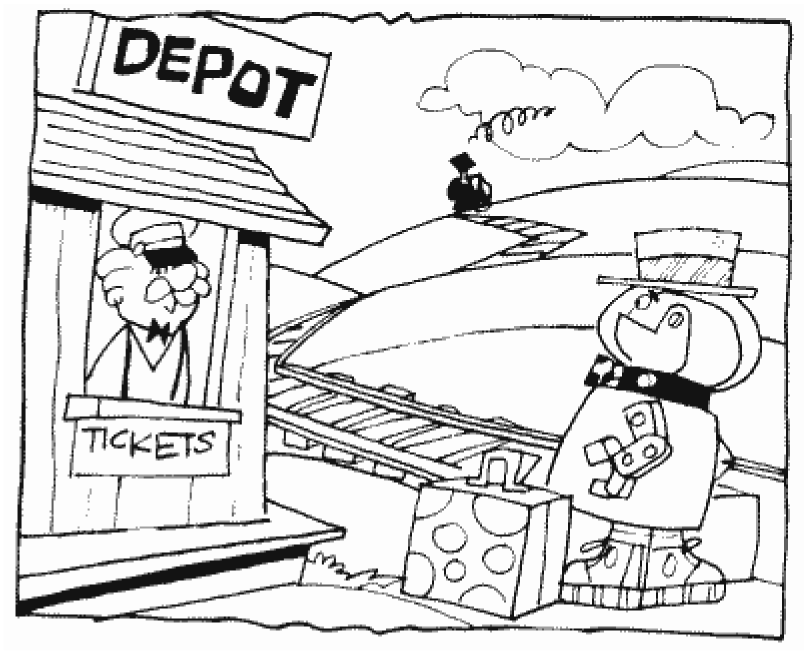
There now, that's much better. Now that we have a spaceship on the screen, though, what can we do with it? Well, the simplest thing to do would be to change its size using SIZEP0 at location 53256. Try the following to see what I mean:

POKE 53256,1
POKE 53256,3
POKE 53256,0

Well, that's nice, but not too exciting. Let's try moving it. We already know how to move it horizontally, so add the following to the program:

```
260 HP0=129
270 S=STICK(0):IF S=9 OR S=10 OR S=11 THEN HP0=HP0-1
280 IF S=5 OR S=6 OR S=7 THEN HP0=HP0+1
290 HP0=HP0-256*(HP0=256)+256*(HP0=-1)
300 POKE 53248,HP0
```

What is going on here? First of all, we use HP0 to remember what the horizontal position of the player is, since HPOSP0 is POKE only. Then we look at joystick zero and move the player to the left or right if the joystick is moved to the left or right. Finally, we check HP0 to make sure that it hasn't gone below 0 or above 255. When you run this, you'll notice that you can move the player off the screen. A nice way to get rid of a player when you don't want it to be on the screen is to set its horizontal position to 0.



That was pretty easy, and it looks pretty good. What about vertical movement though? You may have noticed that there are no vertical position registers. This means that you have to move each of the bytes in the player forward or backward in memory to move the player down and up, respectively. This can be a real pain and is also very slow. So slow, in fact, that I'm not even going to bother giving an example in BASIC. Instead, I'll give you two machine language routines, one for moving the player up, and one for moving it down. Here they are along with an example of how to use them (add the BASIC program lines to the preceding program):

```
10 DIM CMS(36),PU$(43),PD$(26)
90 GOSUB 500
260 HP0=128:VP0=128
310 IF S=6 OR S=10 OR S=14 THEN VP0=VP0-1:X=USR(ADR(PU$),
PM+1024,255)
320 IF S=5 OR S=9 OR S=13 THEN VP0=VP0+1:X=USR(ADR(PD$),
PM+1024,255)
330 VP0=VP0-256*(VP0=256)+256*(VP0=-1)
500 FOR CHAR=1 TO 43
510 READ CODE
520 PU$(CHAR,CHAR)=CHR$(CODE)
530 NEXT CHAR
540 FOR CHAR=1 TO 26
550 READ CODE
560 PD$(CHAR,CHAR)=CHR$(CODE)
570 NEXT CHAR
580 RETURN
800 DATA 104,104,133,204,104,133,203,104,168,104
810 DATA 170,177,203,72,138,72,160,1,177,203,136
820 DATA 145,203,200,200,240,10,192,128,208,243
830 DATA 104,72,201,127,208,237,104,168,104,145,203,96
900 DATA 104,104,133,204,104,133,203,104,104,168
910 DATA 177,203,72,136,177,203,200,145,203,136
920 DATA 208,247,104,145,203,96
```

Both PU$ (Player Up) and PD$ (Player Down) have the same USR format:

X=USR(ADR(PU$ or PD$),*ADDRESS*,*HEIGHT* - 1)

ADDRESS is the starting address of the player you want to move and HEIGHT-1 is 127 for 128 byte players, 255 for 256 byte players.

Here’s the assembly code for Player Up (PU):

```
68    PU     PLA
68           PLA
85CC         STA PLAYERHI
68           PLA
85CB         STA PLAYERLO
68           PLA
A8           TAY
68           PLA
AA           TAX
B1CB         LDA (PLAYERLO),Y
48           PHA
8A           TYA
48           PHA
A001         LDY #1
B1CB  LOOP   LDA (PLAYERLO),Y
88           DEY
91CB         STA (PLAYERLO),Y
C8           INY
C8           INY
F00A         BEQ RETURN
C080         CPY #128
D0F3         BNE LOOP
68           PLA
48           PHA
C97F         CMP #127
D0ED         BNE LOOP
68   RETURN  PLA
A8           TAY
68           PLA
91CB         STA (PLAYERLO),Y
60           RTS
```

Here's the assembly code for Player Down (PD):

```
68    PD    PLA
68          PLA
85CC        STA PLAYERHI
68          PLA
85CB        STA PLAYERLO
68          PLA
68          PLA
A8          TAY
B1CB        LDA (PLAYERLO),Y
48          PHA
88    LOOP  DEY
B1CB        LDA (PLAYERLO),Y
C8          INY
91CB        STA (PLAYERLO),Y
88          DEY
D0F7        BNE LOOP
68          PLA
91CB        STA (PLAYERLO),Y
60          RTS
```

Now we can move our player in any direction. By the way, if you're using 128 byte player/missiles and you want to move them vertically by one scan line instead of two, use VDELAY at location 53276.

Is there still more? You bet. Let's put some background on the screen:

```
121 COLOR 1:PLOT 19,0:DRAWTO 19,23
122 COLOR 2:PLOT 21,0:DRAWTO 21,23
123 COLOR 3:PLOT 20,0:DRAWTO 20,23
```

Nothing great so far, but wait until we fool around with GPRIOR at location 623. Try each of the following and see what happens (see GPRIOR for an explanation of why):

```
195 POKE 623,4
195 POKE 623,8
```

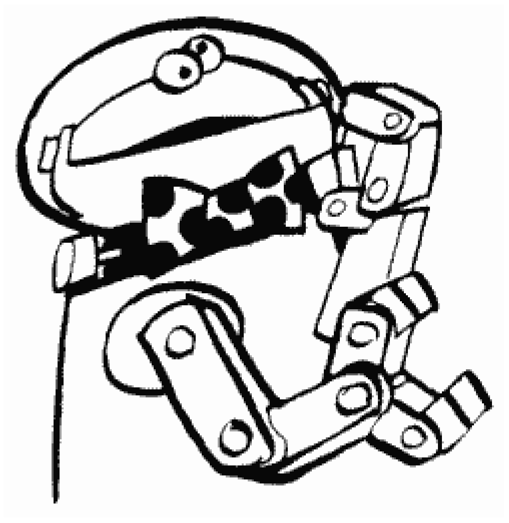
You can also use GPRIOR to mix colors when two players overlap, and to make the four missiles into an extra player. See GPRIOR for more details.

As long as we have background on the screen, let's take a look at the collision registers. P0PF at location 53252 is used to tell whether player zero has collided with any part of the background, or playfield. See P0PF and HITCLR (53278) and try the following:

```
400 POKE 53278,0
410 C=PEEK(20)
420 IF C+1=256 THEN C=-1
430 IF PEEK(20)<C+1 THEN 430
440 SOUND 0,PEEK(53252)*16+15,4,8
450 GOTO 270
```

Lines 410 and 420 are necessary to make sure that GTIA/CTIA has enough time to check for collisions. Note that the collision registers can be used to detect a collision between any two objects on the screen.

Well, we've just about covered everything now with one big exception: the missiles. Missiles are handled in much the same way as players, with several notable exceptions. First of all, they have the same color as the corresponding player, so you would use PCOLR0 to set the color for both player zero and missile zero. Secondly, there is only one size register for all four missiles, although you can set the size of each missile separately (see SIZEM at location 53260). But the most important difference, obviously, is in the way the missiles are stored in memory. Because all four are stored in the same bytes, it can be very difficult to move them vertically, since you must only move two bits for each, without disturbing the others. I'll come to your rescue again, however, and provide you with the machine language routines necessary to do just that:



```
68      MU     PLA
68             PLA
85CC           STA PLAYERHI
68             PLA
85CB           STA PLAYERLO
68             PLA
68             PLA
85CD           STA MASK
68             PLA
A8             TAY
68             PLA
AA             TAX
A5CD           LDA MASK
49FF           EOR #255
31CB           AND (PLAYERLO),Y
48             PHA
8A             TXA
48             PHA
B1CB   LOOP    LDA (PLAYERLO),Y
25CD           AND MASK
91CB           STA (PLAYERLO),Y
C8             INY
A5CD           LDA MASK
49FF           EOR #255
31CB           AND (PLAYERLO),Y
88             DEY
11CB           ORA (PLAYERLO),Y
91CB           STA (PLAYERLO),Y
C8             INY
F00A           BEQ RETURN
C080           CPY #128
D0E7           BNE LOOP
68             PLA
48             PHA
C97F           CMP #127
D0E1           BNE LOOP
68     RETURN  PLA
A8             TAY
A5CD           LDA MASK
31CB           AND (PLAYERLO),Y
91CB           STA (PLAYERLO),Y
68             PLA
11CB           ORA (PLAYERLO),Y
91CB           STA (PLAYERLO),Y
60             RTS
```

```
68    MD     PLA
68           PLA
85CC         STA PLAYERHI
68           PLA
85CB         STA PLAYERLO
68           PLA
68           PLA
85CD         STA MASK
68           PLA
68           PLA
A8           TAY
A5CD         LDA MASK
49FF         EOR #255
31CB         AND (PLAYERLO),Y
48           PHA
B1CB  LOOP   LDA (PLAYERLO),Y
25CD         AND MASK
91CB         STA (PLAYERLO),Y
88           DEY
A5CD         LDA MASK
49FF         FOR #255
31CB         AND (PLAYERLO),Y
C8           INY
11CB         ORA (PLAYERLO),Y
91CB         STA (PLAYERLO),Y
88           DEY
D0EB         BNE LOOP
A5CD         LDA MASK
31CB         AND (PLAYERLO),Y
91CB         STA (PLAYERLO),Y
68           PLA
11CB         ORA (PLAYERLO),Y
91CB         STA (PLAYERLO),Y
60           RTS
```

These routines need the following USR call:

X=USR(ADR(MU$ or MD$),*ADDRESS*,*MASK*,*HEIGHT*-1)

ADDRESS AND HEIGHT-1 are the same as they were for PU$ and PD$, MASK has one of the following values:

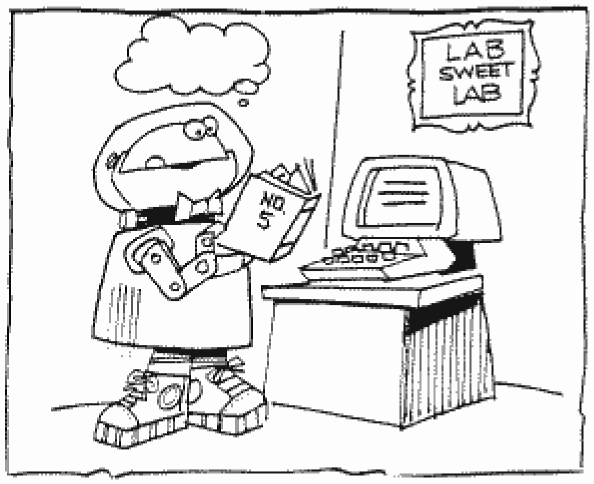
252 if you’re moving missile zero.
243 if you’re moving missile one.
207 if you’re moving missile two.
63 if you’re moving missile three.

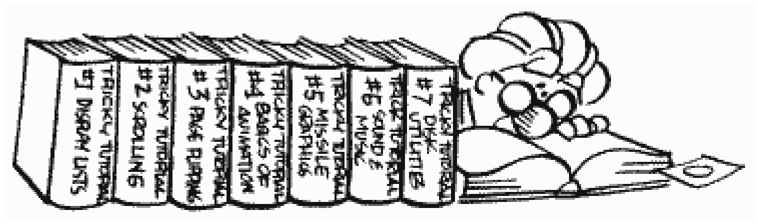
"Wait a minute, aren't you forgetting something? Where's the BASIC routine to set up MU$ and MD$?" Don't worry, it's on the way.

If you make the following changes to the preceding player program, it will move missile zero instead of player zero. Note that even though I'm only putting one thing on the screen at a time here, there is no reason why all four players and missiles can't be used together. I'm just lazy:

```
10 DIM CM$(36),MU$(69),MD$(54)
200 FOR BYTE=127 TO 128
220 POKE PM+768+BYTE,DAT
240 POKE 53252,128
300 POKE 53252,HP0
310 IF S=6 OR S=10 OR S=14 THEN VP0=VP0-1:X=USR(ADR(MU$),
PM+768,252,255)
320 IF S=5 OR S=9 OR S=13 THEN VP0=VP0+1:X=USR(ADR(MD$),
PM+768,252,255)
440 SOUND 0,PEEK(53248)*16+15,4,8
500 FOR CHAR=1 TO 69
520 MU$(CHAR,CHAR)=CHR$(CODE)
540 FOR CHAR=1 TO 54
560 MD$(CHAR,CHAR)=CHR$(CODE)
800 DATA 104,104,133,204,104,133,203,104,104,133
810 DATA 205,104,168,104,170,165,205,73,255,49,203
820 DATA 72,138,72,177,203,37,205,145,203,200,165
830 DATA 205,73,255,49,203,136,17,203,145,203,200
840 DATA 240,10,192,128,208,231,104,72,201,127
850 DATA 208,225,104,168,165,205,49,203,145,203
860 DATA 104,17,203,145,203,96
900 DATA 104,104,133,204,104,133,203,104,104,133
910 DATA 205,104,104,168,165,205,73,255,49,203,72
920 DATA 177,203,37,205,145,203,136,165,205,73,255
930 DATA 49,203,200,17,203,145,203,136,208,235,165
940 DATA 205,49,203,145,203,104,17,203,145,203,96
1000 DATA 255,255
```

These are only the fundamentals of using players and missiles. You will learn best by experimenting with what we have given you. See what happens if you change things. If you want a complete lesson on the subject, Educational Software offers a fun to use program called “Tricky Tutorial™ #5.” Write us for a list of all the tutorials.




TRICKY TUTORIAL #1 DISPLAY LISTS
TRICKY TUTORIAL #2 SCROLLING
TRICKY TUTORIAL #3 PAGE FLIPPING
TRICKY TUTORIAL #4 BASICS OF ANIMATION
TRICKY TUTORIAL #5 MISSILE GRAPHICS
TRICKY TUTORIAL #6 SOUND & MUSIC
TRICKY TUTORIAL #7 DISK UTILITIES

APPENDIX THREE

DESIGNING YOUR OWN GRAPHICS MODES

This appendix assumes that you have already read the description of SDLSTL at locations 560 and 561 and SAVMSC at locations 88 and 89.

Since designing a custom graphics mode is so simple, let's jump right into it. The first step is to decide what you want the new mode to look like. In other words, figure out how you want to mix the existing graphics modes. For our example, let's mix graphics modes zero, one, two, and seven.

So far so good. Next comes the tricky part; we have to decide how many rows of each mode we want. Why is this tricky? Because we have to have 192 scan lines altogether (remember that a scan line is the height of a graphics mode eight line, or row). If we have less than 192, then the display will be too short on the screen. Similarly, too many will make it too long. You can try it later and see for yourself. Anyway, we need to know how many scan lines high each graphics mode row is. Figure 57 tells us just that.

MODE	0	1	2	3	4	5	6	7	8-11
LINES	8	8	16	8	4	4	2	2	1

FIGURE 57. Designing PM graphics modes

For our example, suppose that we definitely want one row of mode two, two of mode one, and three of mode zero. That gives us a total of 16+2*8+3*8 = 56 scan lines, leaving 192-56 = 136 scan lines left over for mode seven. Since mode seven rows are two scan lines high, we can have 68 of them. Oh, I forgot to mention that we're going to want the mode two row on top, followed by half the mode seven rows, the mode one rows, the rest of the mode seven rows, and then the mode zero rows on the bottom.

Now we're ready to make our display list. Since BASIC has a built-in ability to set up a normal display list (and screen memory), let's not waste it. We'll have BASIC set up the mode that uses the most memory out of the ones we want to mix (that way, we know there will be enough screen memory for us). In our case that's graphics mode seven, so we'll start with the following line:

```
100 GRAPHICS 7+16
```

We use 7+16 because we don't want a text window. Now we want to change the first row into graphics mode two. First we need to know where the display list is:

```
110 DLIST=PEEK(560)+PEEK(561)*256
```

We'll also turn off the screen so that funny things don't happen while we're changing things, and put an endless loop at the end so we won't go back to graphics mode zero when we're through:

```
120 POKE 559,0
999 GOTO 999
```

You'll recall from SDLSTL that the first three bytes of a display list are used to keep the first 24 scan lines blank. That means that the fourth byte in the display list is the instruction for the first row. You may also recall that this instruction has an LMS modification to it, since ANTIC has to know where the screen memory is before it can start drawing the screen. So we look at the chart in Appendix Twelve and find that a mode two LMS instruction has a value of 71. So…

```
130 POKE DLIST+3,71
```

In case the DLIST+3 is confusing you, keep in mind that DLIST is the first byte, not DLIST+1.

Now we've got our mode two row. Next we want 34 mode seven rows. But wait a minute, we started with a graphics mode seven display list so that means that the mode seven rows are already set up for us. Great, let's go on to mode one.

Our two mode one rows will be the thirty-sixth and thirty-seventh rows on the screen, right? That corresponds to the forty-first and forty-second bytes in the display list. Why? Don't forget that the first three bytes are the BLK instructions and then we have two more bytes for the LMS address. Looking at Appendix Twelve again, we see that a graphics mode one instruction has a value of six, so we add the following lines to our program:

```
140 POKE DLIST+40,6
150 POKE DLIST+41,6
```

Now we want the rest of our mode seven rows, which again are already set up for us, in DLIST+42 through DLIST+75, so we skip over to mode zero. The seventy-second through seventy-fourth rows will be mode zero, so we'll change the seventy-seventh through seventy-ninth bytes:

```
160 POKE DLIST+76,2
170 POKE DLIST+77,2:POKE DLIST+78,2
```

And now we're done, right? Not quite, we still have to put in the JVB instruction (see SDLSTL):

```
180 POKE DLIST+79,65
190 POKE DLIST+80,PEEK(560)
200 POKE DLIST+81,PEEK(561)
```

Now we're done, so let's turn the screen back on:

```
210 POKE 559,34
```

Uh-oh, we can see the graphics mode zero rows down at the bottom. That's easy enough to change:

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

Much better. Now comes the fun part-trying to put stuff on the screen. There are a lot of problems associated with this, so let's look at them one by one.

We'll start at the top, with our mode two row. What's the problem here? The OS thinks that it's in graphics mode seven, since we told it it was. That's easy to change, however, since DINDEX (87) can be changed to indicate the mode we want:

```
230 POKE 87,2
```

Now we can go ahead and PRINT to the row:

```
240 POSITION 0,0
250 PRINT #6;" graphics mode two"
```

So far so good; let's go on to the first group of mode seven rows. Unfortunately, we can't just change DINDEX back and go ahead and PLOT and DRAWTO. Why not? Graphics mode seven uses 40 bytes per row while graphics mode two only uses 20 (see SAVMSC). This will cause things to be PLOTted in the wrong place. Try the following to see for yourself:

```
260 POKE 87,7
270 COLOR 1
280 PLOT 0,1
```

This causes a pixel to be plotted halfway across the screen instead of in the first column. Not to worry, however, there's an easy solution. We can change SAVMSC to point to the beginning of the mode seven area. By doing that, the upper left corner of the mode seven area will be treated as the beginning of the screen. The only hard thing about doing this is that we have to figure out the memory location of the beginning of the first mode seven row. How do we do that? Well, we know that DLIST+4 and DLIST+5 hold the LMS address, which is the address of the beginning of screen memory. We also know that there is a mode two row between the beginning of screen memory and the beginning of our mode seven rows. From the table at location SAVMSC, we know that a mode two row takes up 20 bytes, so that means the beginning of the mode seven rows is 20 bytes after the beginning of screen memory. All of this together gives us:

```
260 SCRMEM=PEEK(DLIST+4)+PEEK(DLIST+5)*256
270 MEM7A=SCRMEM+20
280 POKE 89,INT(MEM7A/256)
290 POKE 88,MEM7A-PEEK(89)*256
```

Now we can go ahead and PLOT and DRAWTO, remembering that there are only 34 mode seven rows before we hit the mode one rows. Oh, we also have to remember to change DINDEX:

```
300 POKE 87,7
310 COLOR 1
320 PLOT 0,0
330 DRAWTO 159,0:DRAWTO 0,33
340 DRAWTO 159,33:DRAWTO 0,0
```

For our mode one lines we'll follow the same procedure. There are 1 mode two rows and 34 mode seven rows between the beginning of screen memory and the beginning of the mode one rows. That's a grand total of 20+34*40 = 1380 bytes:

```
350 MEM1=SCRMEM+1380
360 POKE 89,INT(MEMl/256)
370 POKE 88,MEM1-PEEK(89)*256
```

And now to PRINT:

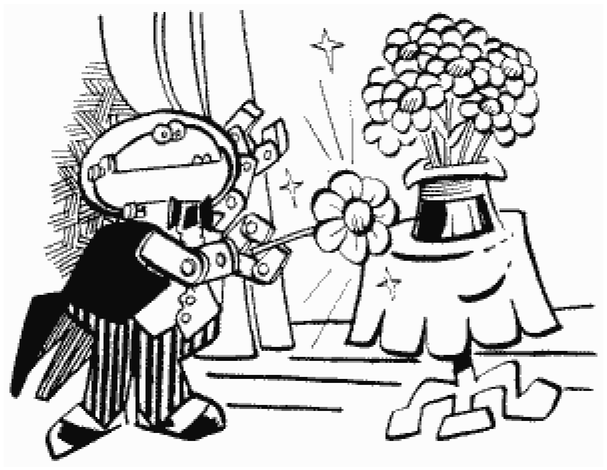
```
380 POKE 87,1
390 POSITION 7,0
400 PRINT #6;"here's"
410 PRINT #6;" graphics mode one"
```

And so it goes for the rest of the screen:

```
420 MEM7B=SCRMEM+1420
430 POKE 89,INT(MEM7B/256)
440 POKE 88,MEM7B-PEEK(89)*256
450 POKE 87,7
460 PLOT 0,0
470 DRAWTO 159,0:DRAWTO 0,33
480 DRAWTO 159,33:DRAWTO 0,0
490 MEM0=SCRMEM+2780
500 POKE 89,INT(MEM0/256)
510 POKE 88,MEM0-PEEK(89)*256
520 POKE 87,0
```

```
530 POKE 752,1
540 POSITION 10,0
550 PRINT #6;"And now, finally..."
560 POSITION 9,1
570 PRINT #6;"...GRAPHICS MODE ZERO"
```

Everything here may seem a little overwhelming at first, but it really isn't. Type in the example and make sure you understand it. Make little changes and see what the effect is. In no time at all you'll be whipping up your own graphics modes at the drop of a hat.



Now that you're confident that you know what's going on, I'll mention the catch. Fortunately, the only time you'll run across it is if you're mixing graphics mode eight into your custom mode. You'll recall that back at SDLSTL I mentioned that the graphics mode eight display list has to have an extra LMS instruction so it can cross a 4K boundary. Well, this means that you have to be careful when you start your custom mode with a mode eight display list. First of all, find out where the LMS is. Usually it's the hundredth byte (DLIST+99), but your program should check first to make sure. Once you know where it is, make sure you don't change the two bytes after it, since they are the LMS address. Next, don't forget about those two bytes when you change instructions that come after the LMS. For example, let's suppose that the LMS is at DLIST+99 and you want to change the hundred-twenty-fifth row. Normally this would be DLIST+129, but because of the LMS address it's now DLIST+ 131. The final thing to watch out for is changing SAVMSC for rows that come after the LMS. What you'll have to do is use the LMS address instead of SCRMEM, and figure out how many bytes past that your mode rows begin. As you can see, this can get to be a real pain in the you-know-what. Unfortunately, there seems to be no way around it.

APPENDIX FOUR

GTIA GRAPHICS MODES NINE, TEN, AND ELEVEN

First of all, I'm going to assume that you've already read the information about GTIA at location GPRIOR (623) and at the GTIA chip (53248). If you haven't, then do so now.

Do I have a GTIA?

All of you should now know what a GTIA chip is, so let's try and figure out a way to tell if your Atari has one. Try the following:

POKE 623,64

If the screen goes black, then you have a GTIA. If it doesn't, well then you have a CTIA. Real simple test, right? But what happens if you're writing a program that has to know whether or not it's running on a GTIA. You can't use the preceding method unless your program asks the user whether the screen went black or not. This is not a good idea. Instead, the program needs a way of finding out what's there itself. How? In graphics modes nine and eleven, the computer can't detect collisions between players/missiles and the playfield. What we do then, is set up a collision and see whether or not it was detected. If it was, then we know that there is a CTIA chip, otherwise there is a GTIA. The following program will do the trick:

```
100 GRAPHICS 9:POKE 710,0
120 POKE 53248,45
130 POKE 53261,255
140 POKE 53278,0
150 COLOR 1:PLOT 0,0
160 TI=PEEK(20)
170 IF TI=255 THEN TI=-1
```

```
180 IF PEEK(20)<TI+1 THEN 180
190 POKE 53248,0:COLID=PEEK(53252)
200 GRAPHICS 0
210 IF COLID=0 THEN PRINT "GTIA":END
220 PRINT "CTIA"
```

So you can understand a little better what's going on here, let's substitute the location names for the addresses:

```
100 GRAPHICS 9:POKE COLOR2,0
120 POKE HPOSP0,45
130 POKE GRAFP0,255
140 POKE HITCLR,0
150 COLOR 1:PLOT 0,0
160 TI=PEEK(RTCLOK+2)
170 IF PEEK(RTCLOK+2)<TI+1 THEN 170
180 POKE HPOSP0,0:COLID=PEEK(P0PF)
190 GRAPHICS 0
200 IF COLID=0 THEN PRINT "GTIA":END
210 PRINT "CTIA":END
```

What does GTIA have to offer me?

OK, so now we know whether or not we have a GTIA. Assuming we do have one, what can we do with it? Well, GTIA gives us three extra graphics modes which we can use to get many more colors on the screen than usual. Graphics mode nine lets you put all 16 brightnesses of one color on the screen at the same time. Graphics mode ten lets you have 9 colors on the screen at the same time (you pick the brightness for each). Finally, graphics mode eleven lets you put 16 colors at the same brightness on the screen at the same time. Now at this point you may be wondering if there are any strings attached to these great new graphics capabilities. Yes! Each mode takes the same amount of memory as graphics mode eight (a little less than 8K), and each pixel is four times as wide as those in graphics mode eight. That means the screen is 80 pixels wide and 192 high. Why four times as wide? Each pixel in graphics mode eight is only one bit wide, since it can only be on or off. In these new modes, however, a pixel can have up to 16 different values depending on what color it is (in case you hadn't guessed, the value of a pixel determines what color it is). You need four bits to get 16 values, so each pixel has to be four bits wide. In graphics mode ten, even though there are only 9 colors, four bits are still needed since three bits would only give you eight values. In this mode, the extra seven values just pick the same colors as the first nine. By the way, if you try to use a GTIA program on a CTIA machine, it will run fine as long as it doesn't use the GTIA modes (with the exception of the artifacting problem). If it uses the GTIA modes, then it will still run, but the graphics will look funny.

How do I use these extra modes?

All of these modes can be set up just the same as the other nine. For example, graphics nine, graphics ten, and graphics eleven all work from BASIC. Note that you can't have a text window in these modes. If you set up your own display list with machine language, you need to set GPRIOR according to the mode you want to use. Also, if you are going to be using the OS PLOT and DRAW routines, make sure you set DINDEX to the correct mode number. Incidentally, as you probably guessed, all three modes have the same screen memory requirements as graphics mode eight.

Now that you've got the mode set up and ready to go, how do you use it? Each one is a little different in this respect, so let's look at each separately:

Graphics Nine

As mentioned before, graphics nine gives you one color with 16 brightnesses. The background color register is used to tell GTIA what color you want to use. From BASIC, you just do the following:

SETCOLOR 4,*color*,0

Or this, which is more pertinent for machine language programmers:

POKE 712,*color**16

To pick the brightness, use the COLOR command before you PLOT or DRAWTO. For example,

COLOR *brightness*

"Brightness" can be any value from 0 to 15, with 15 being the brightest. From machine language, store the brightness value directly into the pixel in screen memory. If you're using the PLOT and DRAW routines, store it in ATACHR (763).

The following BASIC program will set up a graphics mode nine screen and put all 16 brightnesses on it:

```
100 GRAPHICS 9
110 SETCOLOR 4,6,0
120 FOR C=0 TO 15
130 COLOR C
140 FOR ROW=88 TO 103
150 PLOT C*4+ROW-88,ROW
160 DRAWTO C*4+ROW-85,ROW
170 NEXT ROW
180 NEXT C
190 GOTO 190
```

Line 190 is necessary because we don't have a text window on the screen. Take it out and see what happens.

Graphics Ten

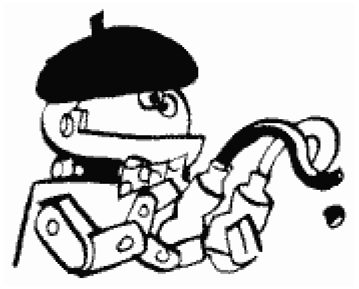
Graphics mode ten only lets you have nine different colors at the same time, but you can have different color and brightness values for each. It uses the regular playfield color registers along with the player/missile color registers to specify these colors. Just POKE the color and brightness values into these registers and you can then use the COLOR command to pick the one you want to use (or POKE the COLOR value directly into screen memory). Figure 58 shows you which COLOR value picks which register.

COLOR	REGISTER	LOCATION
0	PCOLR0	704
1	PCOLR1	705
2	PCOLR2	706
3	PCOLR3	707
4	COLOR0	708
5	COLOR1	709
6	COLOR2	710
7	COLOR3	711
8	COLOR4	712

FIGURE 58. Color register locations

(PCOLR0 no specifies the background color instead of COLOR4.) The following program is an example of using graphics mode ten:

```
100 GRAPHICS 10
110 FOR LP=0 TO 8
120 POKE 704+LP,LP*16+LP*2+10
130 NEXT LP
140 FOR C=1 TO 8
150 COLOR C
160 FOR ROW=C TO 71 STEP 8
170 PLOT ROW,88
180 DRAWTO ROW+6,103
190 NEXT ROW
200 NEXT C
210 T=PEEK(712)
220 FOR REG=712 TO 706 STEP -1
230 POKE REG,PEEK(REG-1)
240 NEXT REG
250 POKE 705,T
260 GOTO 210
```



Graphics Eleven

Graphics mode eleven is the same as graphics mode nine except that we have 16 colors with the same brightness instead of the other way around. To pick the brightness, use the following from BASIC:

SETCOLOR 4,0,*brightness*

Or this:

POKE 712,*brightness*

where brightness has an even value between 0 and 14.

To pick a color, use:

COLOR *C*

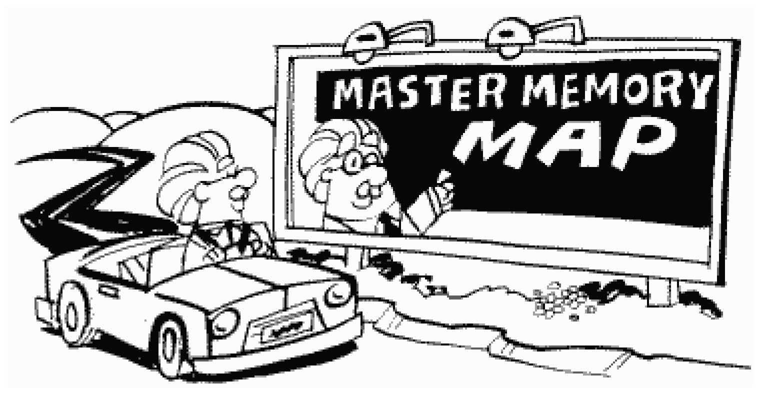
where "C" has a value between 0 and 15. Machine language programmers should just store the color value in either the pixel or in ATACHR, as for the brightness value in graphics mode nine.

Try the following changes to the graphics mode nine program:

```
100 GRAPHICS 11
110 SETCOLOR 4,0,6
120 FOR C=0 TO 15
130 COLOR C
140 FOR ROW=88 TO 103
150 PLOT C*4+ROW-88,ROW
160 DRAWTO C*4+ROW-85,ROW
170 NEXT ROW
180 NEXT C
190 GOTO 190
```

As you can see, modes nine and eleven really are almost the same.

One final note: You can still use players and missiles in graphics modes nine and eleven for a grand total of 21 different colors on the screen at one time! You will not, however, be able to detect collisions between players/missiles and playfield.



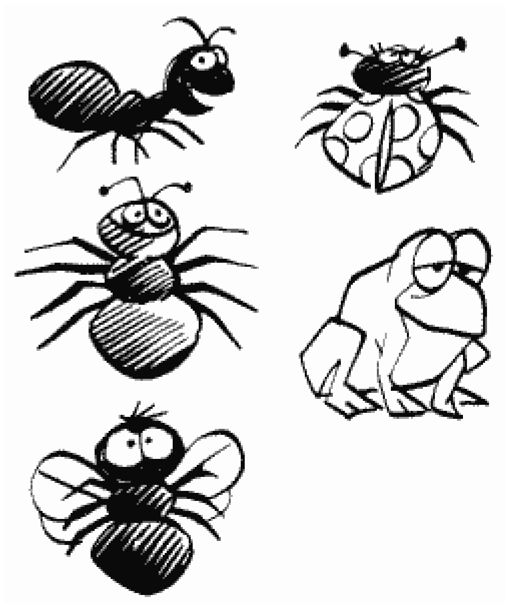
APPENDIX FIVE
THE DIFFERENT VERSIONS OF THE OS

As has been mentioned throughout the book, there are currently two versions of the OS (three, actually, if you count the 1200XL). The reason for this is the fact that the original version of the OS, which we'll call version "A," had some bugs, or problems that caused it not to work properly sometimes. Version "B" was developed to take care of these bugs, but in the process a lot of the routines got moved. As a result, some of the **OS locations** listed in this book will only be correct for the version "A" OS. This will only affect you, however, if you are using the OS incorrectly. What do I mean by incorrectly? When the Atari computers first came out, Atari warned that programmers should not rely on the OS locations staying the way they were. If you wanted to use the OS in your own programs, said Atari, make sure you only use locations that are safe, mainly the vectors . As a matter of fact, the whole purpose of OS vectors is to let you use routines that might shift around. Anyway, my point here is that you should not try and use an OS routine if it doesn't have a vector. If you do, then don't expect your program to run on every Atari. If you absolutely must use a nonvectored OS routine, look it up in the OS listing and copy it into your program. The only problem with this method is that it may create legal problems if you decide to sell your program commercially. Check with Atari in such cases.

If you're stubborn, and insist on making "illegal entries" into the OS, here's a list of the locations that are **already** unsafe (i.e., they've been changed):

58460 through 58466 ($E45C through $E462)
59126 through 60905 ($E6F6 through $EDE9)
62015 through 62158 ($F23F through $F2CE)

This represents a little less than a fifth of the OS. You should be aware, however, that not **all** the locations in the given ranges have been changed, and some of the routines have just been moved. Since it is **highly** recommended that you not enter the OS illegally, however, I'm not going to go into any more depth.



In case you were wondering about the bugs that version “B” fixed, here’s a list:

1. Sometimes, during I/O to the disk drive, the disk drive would “fall asleep” (timeout) for a few seconds.

2. Sometimes the screen display would disappear.

3. Sometimes you would get an incorrect “ERROR 138” message during I/O (ERROR 138 is a device timeout error).

4. POKEY timer four did not work properly.

5. Sending SIO a buffer address with a low byte of 255 ($FF) caused SIO to loop forever.

6. There was no vector for the BREAK key.

APPENDIX SIX

BASIC BUGS

That’s right, even BASIC has problems. As of this writing, the BASIC cartridge suffers from the following ailments:

1. If you use INPUT without a variable, BASIC won’t give you an error message, but will instead lock up when it reaches the INPUT. This is fixed in Rev. C.

2. Sometimes when you’re making a lot of changes to a program (especially if you’re deleting lines or pressing RETURN a lot), BASIC will suddenly stop talking to you and won’t let you see your program again.

3. BASIC has problems with strings that are DIMensioned to a multiple of 256 (regardless of whether you use all 256 characters or not). If you have to have a string that’s a multiple of 256 long, add one more when you DIMension it.

4. If for some reason you need to use the statement “PRINT A=NOT B”, don’t; it will put the computer to sleep. This is fixed in Rev. C.

5. The cassette handler doesn’t always set itself up properly when you tell BASIC to CSAVE or SAVE "C:". To get around this, and thus make sure your program is saved properly, type “LPRINT” and press RETURN before you save the program (make sure you printer is off if you have one). This will give you an error message, which you can just ignore.

6. Strange things can happen if you type a program line that’s longer than three screen lines.

7. A lot of the mathematics is slightly off. For example, five cubed (5^3) will equal 124.999998 instead of 125. Some other functions do this also. You may want to round up if you need the accuracy. This is fixed in Rev. C.

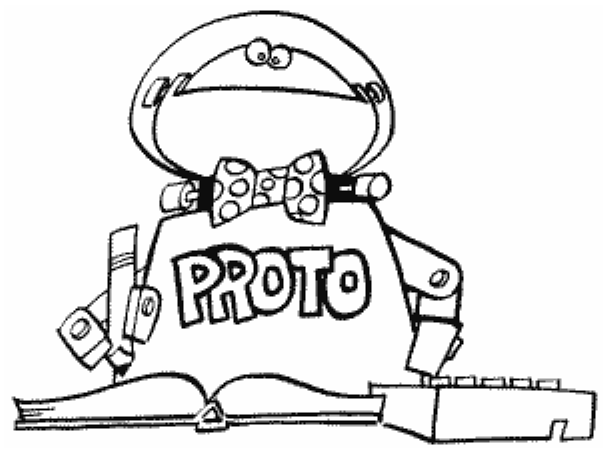
8. A printed CTRL-R or CTRL-U is treated like a semi-colon. This is fixed in Rev. C.

9. BASIC does not like variable names that begin with “NOT”. It get confused and changes the variable name to NOT*varname*. Example: NOTHING will be changed to NOT thing.

10. LOCATE and GET don’t initialize properly, which means that they may give you different results when you RUN your program a second time. If you use these commands, trick BASIC into initializing them by using a command like A=STR$(0) before the LOCATE or GET (you can use any variable instead of the “A”).

11. If you are going to be inputting more than 128 bytes at a time, then page six in memory will not be safe.

Although rumors abound that Atari is going to release a new version of BASIC to correct these BUGS, they had not done so as of the end of 1983. If you have an XL computer with built-in BASIC, try these to see if they have been fixed.



APPENDIX SEVEN

INPUT/OUTPUT

Input/Output, or I/O as it is more commonly called, is an extremely important part of any computer. Without it, the various parts of the computer wouldn't be able to talk to each other. Such communication isn't limited to disk drives and printers, either. The keyboard, television set, and screen editor must all be able to tell the computer what's going on, and all of this is I/O's responsibility. Unfortunately, because I/O has so much to do, it can be a little complicated. Luckily, complicated doesn't mean difficult in this case, so despite all the memory locations that deal with I/O, the basic concept is relatively simple.

There are three main routines that take care of I/O for a given device ("device" is just a fancy word for the keyboard, printer, or whatever it is we want to talk to). They are shown in Figure 59a.

Central Input/Output Routine	(CIO)
Device Handler	
Serial Input/Output Routine	(SIO)

FIGURE 59a. I/O devices

To help these three routines talk to **each other** (I told you this got complicated), there are also four "control blocks" (Figure 59b).

Input/Output Control Block	(IOCB)
Zero-page I/O Control Block	(ZIOCB)
Device Control Block	(DCB)
Command Frame Buffer	(CFB)

FIGURE 59b. I/O control blocks

Now that we've got the names straight, let's look at what each does, where it can be found, and how everything is tied together.

IOCB

Actually, it should be IOCBs, since there are eight of them. The IOCBs are found starting at location 832. Each one is made up of 16 bytes that are used to describe what kind of I/O we want to do. Although we can have information in all eight IOCBs at once, we can still only do one I/O operation at a time. IOCBs usually get their information from the user.

ZIOCB

There is only one ZIOCB, starting at location 32. It is set up exactly like an IOCB, and as a matter of fact contains the information for the IOCB that is currently being used. Why is it necessary? Because page zero is faster than regular memory, and we want to do I/O as quickly as possible. Since only one IOCB can be used at any one time, it would be a waste of precious page zero memory to put all eight IOCBs in page zero. CIO transfers the information from the IOCB to the ZIOCB.

CIO

The CIO routine can be found in the OS ROM, starting at location 58534. CIO takes the information in the IOCB that is currently being used and stores it in the ZIOCB. It then uses that information along with HATABS (794) to figure out which device handler is needed and then passes control to the device handler.

Device Handler

Again, it should be device handlers, since there is one for each device. The device handlers can be anywhere in memory, with HATABS containing a list of where to find them. A device handler does one of two things. If the device in question is the keyboard, the screen, or the screen editor, then the device handler takes care of the I/O itself. If it's something like the disk drive or printer that's plugged into the computer, then the device handler sets up the DCB with the information it needs.

The exception to this is the disk interface routine, which is also known as the internal disk handler. If you're not using DOS, then you have to set up the DCB yourself in order to talk to the disk drive. If you are using DOS, then a regular disk handler has been loaded into the computer and you can communicate with the disk drive through the IOCBS.

DCB

The DCB (there is only one) is found starting at location 768. It is 12 bytes long and is sort of like the IOCBs in the respect that it holds information that describes what kind of I/O we want to do. This time, however, the information is for SIO instead of CIO.

SIO

The SIO routines, like the CIO ones, are in the OS ROM, starting at location 59716. SIO takes the information in the DCB and uses it to talk to devices that use serial I/O (printer, disk drive, cassette player, etc.). It also sets up and uses the CFB.

CFB

Last of all, we have the CFB. Made up of four bytes starting at location 570, it helps SIO talk to the devices. The CFB is the one part of the I/O system that you should not mess around with yourself.

Let's summarize by looking at the flow of information. The user sets up an IOCB and calls CIO. CIO takes the information in the IOCB, transfers it to the ZIOCB, figures out what device handler is needed, and transfers control to that handler. From there, the handler takes care of internal devices, or sets up the DCB and calls SIO if we need to communicate with a serial (external) device. Finally, SIO takes the information in the DCB, sets up the CFB, and does the I/O. After the I/O has been completed, whether by the handler or SIO, control is transferred back to the user. As a programmer, you can skip any of these steps with the exception of SIO. You should not skip all the way to the CFB.

BASIC programmers may want to look at the excellent article in issue 13 of Analog Computing for ways to go straight to CIO.

APPENDIX EIGHT

IOCB COMMAND BYTE VALUES

The third byte in each IOCB is the command byte, called IOCMD. This is the byte that tells CIO what kind of I/O we want to do (see Appendix Seven if none of this makes sense). It can have the values in Figure 60.

VALUE	MEANING	BASIC equivalent
3	OPEN channel	OPEN #n
5	GET text record (line)	INPUT #n
7	GET bytes (buffer)	GET #n
9	PUT text record (line)	none
11	PUT bytes (buffer)	PUT #n
12	CLOSE channel	CLOSE #n
13	GET status	none

FIGURE 60. IOCB command byte values

In case you're wondering what happened to PRINT #n, BASIC uses a special vector to talk directly to the handler for this particular statement.

You should note that with the "GET bytes" and "PUT bytes" routines, BASIC only allows you to GET and PUT one byte at a time. If you access CIO directly, however, you can GET or PUT a whole buffer. With this in mind, you may ask what the difference is between GETting or PUTting a buffer and GETting of PUTting a record. If there are no carriage returns (End-Of-Lines or EOLs) in the buffer then there is no difference. A record, however, ends when either the end of the buffer is reached or an EOL is encountered.

CIO takes care of all of these commands. Some of the devices also have their own special commands, which the corresponding device handler takes care of. Here's a list of those commands, which you can access using BASIC's XIO command:

Display Handler

17	DRAW line
18	FILL

Disk File Manager

32	RENAME file
33	DELETE file
35	LOCK file
36	UNLOCK file
37	POINT
38	NOTE
254	FORMAT disk

RS-232 Handler

32	Force short block
34	CONTROL DTR, RTS, XMT
36	Configure baud rate
38	Configure translation mode
40	Start concurrent I/O mode

For more information on any of these commands, you should see the OS manual or the 850 Interface manual.

As mentioned in Appendix Seven, if you want to use the resident disk handler (i.e., you're not using DOS or FMS at all), you have to set up the DCB and then do a JSR DISKINV (58451). The DCB has different command values than the IOCB, of course, and a list can be found under DCOMND at location 770.

APPENDIX NINE

CHARACTER VALUES

Most of the characters with values between 128 and 255 are just the inverse characters of the ones with values between 0 and 127. In other words, add 128 to a character value and you get that character in inverse video, with the following exceptions:

Character	ATASCII
RETURN	155
Shift-DELETE	156
Shift-INSERT	157
Ctrl-TAB	158
Shift-TAB	159
Ctrl-2	253
Ctrl-DELETE	254
Ctrl-INSERT	255

Character	Internal	ATASCII	Screen Representation
space	0	32	
!	1	33	!
"	2	34	"
#	3	35	#
$	4	36	$
%	5	37	%
&	6	38	&
:	7	39	'
(	8	40	(
)	9	41	)
*	10	42	*
+	11	43	+
,	12	44	,
-	13	45	-
.	14	46	.
/	15	47	/
0	16	48	0
1	17	49	1
2	18	50	2
3	19	51	3
4	20	52	4
4	21	53	5
6	22	54	6

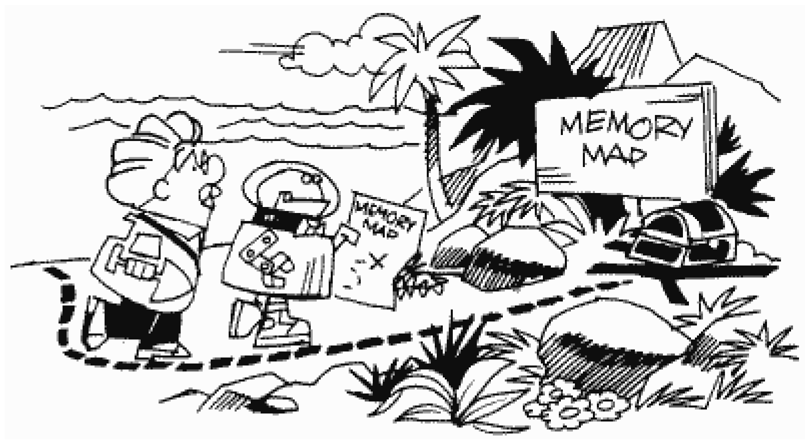
Character	Internal	ATASCII	Screen Representation
7	23	55	7
8	24	56	8
9	25	57	9
:	26	58	:
;	27	59	;
<	28	60	<
=	29	61	=
>	30	62	>
?	31	63	?
@	32	64	@
A	33	65	A
B	34	66	B
C	35	67	C
D	36	68	D
E	37	69	E
F	38	70	F
G	39	71	G
H	40	72	H
I	41	73	I
J	42	74	J
K	43	75	K
L	44	76	L
M	45	77	M

Character	Internal	ATASCII	Screen Representation
N	46	78	N
O	47	79	O
P	48	80	P
Q	49	81	Q
R	50	82	R
S	51	83	S
T	52	84	T
U	53	85	U
V	54	86	V
W	55	87	W
X	56	88	X
Y	57	89	Y
Z	58	90	Z
[	59	91	[
\	60	92	£
]	61	93	]
↑	62	94	↑
←	63	95	←
CTRL-,	64	0	♥
CTRL-A	65	1	├
CTRL-B	66	2	▕
CTRL-C	67	3	┘
CTRL-D	68	4	┤

Character	Internal	ATASCII	Screen Representation
CTRL-E	69	5	┐
CTRL-F	70	6	╱
CTRL-G	71	7	╲
CTRL-H	72	8	◢
CTRL-I	73	9	▗
CTRL-J	74	10	◣
CTRL-K	75	11	▝
CTRL-L	76	12	▘
CTRL-M	77	13	▔
CTRL-N	78	14	▁
CTRL-O	79	15	▖
CTRL-P	80	16	♣
CTRL-Q	81	17	┌
CTRL-R	82	18	─
CTRL-S	83	19	┼
CTRL-T	84	20	●
CTRL-U	85	21	▄
CTRL-V	86	22	▎
CTRL-W	87	23	┬
CTRL-X	88	24	┴
CTRL-Y	89	25	▌
CTRL-Z	90	26	└
ESC	91	27	␛

Character	Internal	ATASCII	Screen Representation
UP ARROW	93	28	⬆
DOWN ARROW	93	29	⬇
LEFT ARROW	94	30	⬅
RIGHT ARROW	95	31	➡
CTRL-.	96	96	♦
a	97	97	a
b	98	98	b
c	99	99	c
d	100	100	d
e	101	101	e
f	102	102	f
g	103	103	g
h	104	104	h
i	105	105	i
j	106	106	j
k	107	107	k
l	108	108	l
m	109	109	m
n	110	110	n
o	111	111	o
p	112	112	p
q	113	113	q
r	114	114	r

Character	Internal	ATASCII	Screen Representation
s	115	115	s
t	116	116	t
u	117	117	u
v	118	118	v
w	119	119	w
x	120	120	x
y	121	121	y
z	122	122	z
CRTL-;	123	123	♠
SHIFT-=	124	124	|
CLEAR	125	125	↖
DELETE	126	126	◀
TAB	127	127	▶



APPENDIX TEN

STAGE TWO VBLANK

At location VVBLKI (546,547), we discussed the various stages of VBLANK and I promised a list of things that were done during stage two. This is that list.

The following hardware registers are updated with the values in their shadow registers (listed in the order they are updated):

Shadow	**Hardware**
LPENV	PENV
LPENH	PENH
SDLSTH	DLISTH
SDLSTL	DLISTL
SDMCTL	DMACTL
GPRIOR	PRIOR
PCOLR0-3	COLPM0-3 (Colors are first adjusted for the
COLOR0-3	COLPF0-3 attract mode.)
COLOR4	COLBK
CHBAS	CHBASE
CHACT	CHACTL
STRIG0-3	TRIG0-3
PADDL0-7	POT0-7

The following shadow registers are updated with the values in the corresponding hardware registers:

Shadow	**Hardware**
STICK0,1	PORTA
STICK2,3	PORTB
PTRIG0-3	PORTA
PTRIG4-7	PORTB

In addition to all this, the following timers are decremented, and the appropriate actions taken if they reach zero:

CDTMV2-5
KEYDEL
SRTIMR

Note that the last two timers deal with the keyboard debounce and repeat, respectively, and are only decremented if necessary. CDTMV1 is taken care of in stage one.

Finally, the last thing stage two does is jump through VVBLKD (548,549).

APPENDIX ELEVEN

THE ATARI XL COMPUTERS

This book was finished just as the 600, 800, & 1400 XL computers were about to be released. Atari has been late in releasing information about the new computers, but our secret "mole," DEEP CHIP, reports that the new computers use the same OS. Experiment to be sure and we will update this section in a future edition. We also have been told that customer service will provide a disk that can cause an XL computer to act like a 400/800. Call them if you have trouble running older programs on your computer.

You may already be aware that there is a big difference between the 1200XL and the earlier Ataris. Not only are there more features, but a lot of the software written for the 400/800 won't run on the 1200XL. Why not? Well, a lot of changes were made to the operating system, and all the people who used parts of the OS other than the vectors found that their programs didn't work on the 1200XL. Even Atari was guilty. So before we take a look at where the differences are, let me once again emphasize that you should never, never, never use any part of the OS other than the safe vectors (see Appendix Thirteen). This means that you should stay away from the OS RAM locations, with the exception of locations that the average person may want to use. This means that locations like LMARGN (82) and CRSINH (752) are safe, but locations like PBUFSZ (30) and FREQ (64) are not. The BASIC manual has a partial list of "safe" locations, with the exception of NEWROW and NEWCOL (96 through 98). If in doubt about whether or not you can use a location, don't. If you absolutely have to, get hold of both types of computers and make sure it works on both.

1200XL OS MEMORY MAP

Before we get into the specific changes to the OS, the following will give you an idea of where things are in the 1200XL memory. It assumes that DOS 2.OS is loaded, and that there is 48K memory. You should be able to make the appropriate changes for no DOS or less memory (Figure 61).

0- 127		0000-007F	OS page zero RAM
128- 255		0080-00FF	BASIC or user page zero RAM
256- 511		0100-01FF	6502 stack
512- 1535		0200-05FF	OS RAM
1536- 1791		0600-06FF	Free RAM
1792- 7419		0700-1CFB	DOS
7420-40959		1CFC-9FFF	BASIC or user RAM
40960-49151		A000-BFFF	Cartridge
49152-52223		C000-CBFF	OS ROM
52224-53247		CC00-CFFF	International character set ROM
53248-53503		D000-D0FF	GTIA
53504-53759		D100-DlFF	Currently unused
53760-54015		D200-D2FF	POKEY
54016-54271		D300-D3FF	PIA
54272-54527		D400-D4FF	ANTIC
54528-55295		D500-D7FF	Currently unused
55296-57343		D800-DFFF	Floating point package
57344-58367		E000-E3FF	Regular character set
58368-65535		E400-FFFF	OS ROM

FIGURE 61. 1200XL memory locations

1200XL OS RAM DIFFERENCES

This section will summarize the differences between the version "B" OS and the 1200XL OS. I apologize that the explanations are not more thorough; detailed information on the 1200XL was not easy to come across at the time of this writing.

LINFLG
0 0000

Used while originally debugging the OS.

NGFLAG
1 0001

A flag for the power up self-test routine. When you first turn on the 1200XL, it checks all its memory locations to make sure they're working correctly.

ABUFPT
28-31 001C-001F

These locations are simply described as "reserved," but from the name I suspect they are some kind of buffer pointers.

PTIMOT, which used to be at location 28, has been moved to 788.

PBPNT has been moved to 734.

PBUFSZ has been moved to 735.

PTEMP no longer exists.

LTEMP
54,55   0036,0037

This, along with any other locations that have the word "loader" in their description, is used by the relocating loader that the 1200XL uses to upload device handlers through the serial I/O port. As this description implies, it's not really something you need worry about.

CRETRY has been moved to 668.

DRETRY has been moved to 701.

ZCHAIN
74,75   004A,004B

A temporary handler loader location

CKEY has been moved to 1001

CASSBT has been moved to 1002.

FKDEF
96,97   0060,0061

There are four function keys on the 1200XL, labeled F1, F2, F3, and F4. FKDEF points to a table of values for the keys. When you press F2, for example, the second value in the table is used as the ATASCII value for the key. There are eight bytes in the table all together, with the last four being for SHIFT-F1, SHIFT-F2, SHIFT-F3, and SHIFTF4. You can, if you want, change FKDEF to point to your own table.

NEWROW has been moved to 757.

NEWCOL has been moved to 758.

PALNTS
98   0062

If PALNTS equals zero, then this is an NTSC Atari. Otherwise it's a PAL. See PAL at location 53268 for an explanation of the two.

NEWCOL+ 1 has been moved to 759.

KEYDEF
121,122 0079,007A

In the 1200XL, not only can you redefine the function keys, you can also define the whole keyboard with the help of KEYDEF. Well, almost the whole keyboard. KEYDEF is of no help if you want to redefine the following:

BREAK, SHIFT, CTRL, OPTION, SELECT, START, RESET, HELP, CTRL-1, CTRL-F1, CTRL-F2, CTRL-F3, CTRL-F4.

What KEYDEF does is point to a 192-byte table. This table contains the ATASCII values to be assigned to the various keys. Here's how the bytes are used:

0- 63	Key alone
64-127	SHIFT plus key
128-191	CTRL plus key

And here's how the keys are ordered within each group of 64 bytes (Figure 62).

	0	1	2	3	4	5	6	7
0	L	J	;	F1	F2	K	+	*
8	0		P	U	RET	I	-	=
16	V	HLP	C	F3	F4	B	X	Z
24	4		3	6	ESC	5	2	1
32	,	SPC	.	N		M	/	INV
40	R		E	Y	TAB	T	W	Q
48	9		0	7	BS	8	<	>
56	F	H	D		CAP	G	S	A

FIGURE 62. 1200XL keyboard layout

Suppose, for example, you pressed the “M” key. “M” is in the column marked “5” and the row marked “32”, which means it is byte 37 (32+5). SHIFT-M would be the 101st byte (37+64), and CTRL-M would be byte 165 (37+ 128). These are the three bytes you would use to store the ATASCII values that you want assigned to those key combinations.

Before you attempt to change the table, you may want to check the one currently being used to get an idea of which values are assigned to each key. This will help you with keys like CTRL-CAPS and so forth. Once you’ve set up all 192 bytes somewhere in memory, change KEYDEF to point to the table.

ROWINC has been moved to location 760.

COLINC has been moved to location 761.

LCOUNT
563 0233

Another temporary loader variable.

RELADR
568,569 0238,0239

A loader variable.

RECLEN
581 0245

Another loader variable.

Noname
583-618 0247-026A

Reserved for undocumented uses.

LINBUF no longer exists.

CHSALT
619 026B

The 1200XL has two built-in character sets: the regular one and a new international one. CHSALT supposedly holds the address (in pages) of the one that is not being used at the moment. It does not, however, seem to have any function at all on the 1200XLs I’ve been using.

You can use CTRL-F4 to switch between the two character sets.

VSFLG
620 026C

This is a temporary fine-scroll location.

KEYDIS
621 026D

If this location is set to 255, then the entire keyboard is disabled, with the exception of the following:

CTRL-F1, RESET, OPTION, SELECT, START

CTRL-F1 will restore KEYDIS to zero, which will re-enable the keyboard. POKEing KEYDIS with any value between 1 and 254 will disable CTRL-F1 as well. The only way to re-enable the keyboard in this case is to POKE KEYDIS with a 0 or press the RESET key.

FINE
622 026E

If you ever wished that you could fine-scroll program listings, or any other output to a graphics mode zero screen, FINE is the location for you. Set it to any value other than zero before your GRAPHICS command, and then try listing a program or PRINTing a lot of stuff to the screen. To get back to regular scrolling, POKE it with a zero before a GRAPHICS command.

HIBYTE
648 0288

A loader location.

CSTAT no longer exists.

NEWADR
654 028E

Another loader location.

CRETRY
668 029C

This was moved here from location 54.
TMPX1 no longer exists.

DRETRY
701 02BD

This was moved here from location 55. HOLD5 no longer exists.

RUNADR
713,714 02C9,02CA

Loader location.

HIUSED
715,716 02CB,02CC

Loader location.

ZHIUSE
717,718 02CD,02CE

Loader location.

GBYTEA
719,720 02CF,02D0

Loader location.

LOADAD
721,722 02D1,02D2

Loader location.

ZLOADA
723,724 02D3,02D4

Loader location.

DSCTLN
725,726 02D5,02D6

Finally, something other than a loader location! DSCTLN is used to specify the length of a disk sector. Power up and RESET set it to 128, but you can set it to anything between 1 and 65536.

ACMISR
727,728 02D7,02D8

Reserved for undocumented uses.

KRPDEL
729 02D9

This one is actually useful. If you press a key and hold it down, after a short delay it will start repeating. KRPDEL specifies the delay. It's initialized to 48, which represents 48/60 = 0.8 seconds. You can change it to any value between 1 and 255, resulting in a delay between one-sixtieth of a second and 4.25 seconds. You can also set it to 0, which will disable the repeat function altogether.

KEYREP
730 02DA

Once a key starts repeating, there is another delay between repeats. KEYREP specifies this delay. It's initialized to six, or one-tenth of a second, which means that you will get 10 characters a second once a key starts repeating. You can make this as fast as 60 characters a second (KEYREP =1) or as slow as 1 character every 4.25 seconds (KEYREP = 255).

NOCLIK
731 02DB

If NOCLIK is set to 0, then the television speaker will make a “clicking” sound every time you press a key. Set it to 255 and it won’t. CTRL-F3 will switch it back and forth between the two values.

By the way, if you set NOCLIK to a value other than 0 or 255, the click will be turned off and CTRL-F3 will have no effect on it; it can only be turned back on by pressing RESET or POKEing NOCLIK with a 0.

HELPFG
732 02DC

HELPFG is used to tell whether or not the HELP key is pressed. It can have the following values:

0	means it’s not pressed.
17	means that HELP is pressed.
81	means that SHIFT-HELP is pressed.
145	means that CTRL-HELP is pressed.

You should note that HELPFG will not change to zero when the HELP key is released. That means that you should POKE 732,0 before you try and read it. Otherwise you may get an old value.

DMASAV
733 02DD

You can use CTRL-F2 to turn off DMA in order to speed up calculations. What happens is the OS stores a zero into SDMCTL at location 559. When you press any other key, SDMCTL will be reset to whatever value it was before. The OS uses DMASAV to hold the value of SDMCTL so it knows what to reset it to.

PBPNT
734   02DE

This was moved here from location 29.

PBUFSZ
735   02DF

This was moved here from location 30.

HNDLOD
745   02E9

A handler loader flag.

CHBAS
756   02F4

Don't worry, this is still the same as it was before. The reason I'm mentioning it here is because there is now a new character set built into the computer. If you POKE CHBAS with a value of 204, you'll get an international character set. This is the same as the regular character set, except that the graphics characters have been redefined and are now letters and characters that English doesn't normally use.

You can use CTRL-F4 to switch back and forth between the two built-in character sets. If you have your own character set in memory, however, pressing CTRL-F4 the first time will switch you to the international character set, but pressing it again will give you the regular character set instead of yours. In other words, you can't use CTRLF4 to switch between the regular character set and your special character set.

NEWROW
757   02F5

This was moved here from location 96.

NEWCOL
758,759   02F6,02F7

This was moved here from locations 97 and 98.

ROWINC
760   02F8

This was moved here from location 121.

COLINC
761	02F9

This was moved here from location 122.

JMPERS
782	030E

JMPERS is used to tell how the system is configured. It is unclear at this time exactly what that means.

PTIMOT
788	0314

This was moved here from location 28.

TEMP2 was moved to location 787.

PUPBT1-PUPBT3
829-831	033D-033F

These are simply labeled as “power-up/reset.” They seem to have some kind of effect on what happens if you press RESET, but I’m not sure what it is.

SUPERF
1000	03E8

Another mystery, this one is simply described as “screen editor.” I assume it’s used as some kind of variable by the screen editor. POKEing values to it doesn’t seem to have any effect, and it always reads zero.

CKEY
1001   03E9

This was moved here from location 74.

CASSBT
1002   03EA

This was moved here from location 75.

CARTCK
1003   03EB

CARTCK is the checksum value for the cartridge. It is used to make sure the cartridge is being read correctly.

ACMVAR
1005-1016   03ED-03F8

Reserved for undocumented uses.

MINTLK
1017   03F9

Also reserved.

CINTLK
1018   03FA

Cartridge interlock. The 1200XL lets you unplug and plug in a cartridge without turning the computer off first. CINTLK tells the OS whether or not a cartridge is currently being plugged in or pulled out. If it's equal to one, then everything is OK. If not, it will reboot the system. That means that POKEing CINTLK with anything other than one will have the same effect as turning the computer off and then back on again.

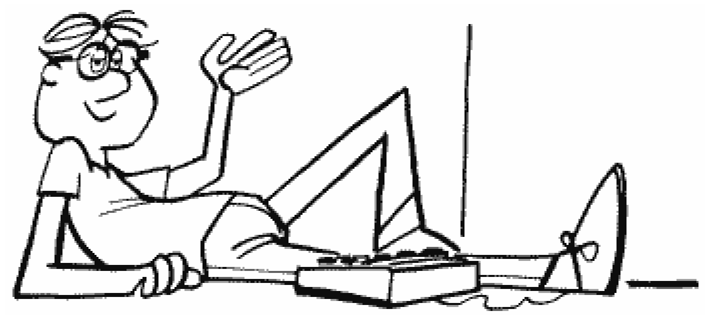
CHUNK
1019,1020   03FB,03FC

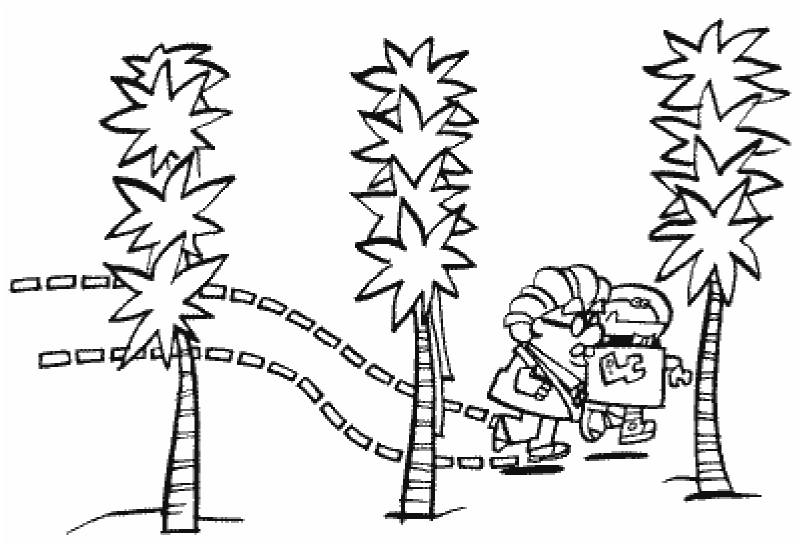
Handler chain. Sorry, but that's all I know about it.

You should also be aware that PORTB in the PIA chip (54017) is no longer used for controller jacks three and four, since there are only two controller jacks on a 1200XL. Similarly, any locations dealing with joystick two, joystick three, paddle four, paddle five, paddle six, or paddle seven will no longer be of any use.

EXTRA GRAPHICS MODES

That's right, the 1200XL gives you four extra modes that the 400/800 doesn't. Actually, that's not quite true. Back at SDLSTL we saw that there were five ANTIC graphics modes that BASIC didn't allow you to use unless you set up your own display list. All the 1200XL does is give the screen handler the capability to set up four of these modes for you. Graphics 12 will give you an ANTIC mode four display list, Graphics 13 ANTIC mode five, Graphics 14 ANTIC mode twelve, and GRAPHICS 15 ANTIC mode fourteen. For more information on each of these modes, see Appendix Twelve on ANTIC modes.





APPENDIX TWELVE

DISPLAY LIST COMMANDS AND ANTIC MODES

As we saw at SDLSTL (560), the display list is actually a little program for ANTIC, telling it what the screen is to look like. It has its own special commands, as summarized in Figure 63. To use this chart, decide which of the HSC, VSC, LMS, and DLI options you want (see SDLSTL for a description of each), find the column that has the corresponding boxes LABELED in at the top, and then follow it down to the instruction you want.

Since this is a chart that assembly language programmers are going to use a lot, I'll also include a hexadecimal version (Figure 64).

			HSC		HSC		HSC		HSC		HSC		HSC		HSC	HSC	
				VSC	VSC			VSC	VSC			VSC	VSC			VSC	VSC
						LMS	LMS	LMS	LMS					LMS	LMS	LMS	LMS
										DLI	DLI	DLI	DLI	DLI	DLI	DLI	DLI
BLK	1	0								128							
BLK	2	16								144							
BLK	3	32								160							
BLK	4	48								176							
BLK	5	64								192							
BLK	6	80								208							
BLK	7	96								224							
BLK	8	112								240							
JMP		1								129							
JVB		65								193							
CHR	2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
CHR	3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
CHR	4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
CHR	5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
CHR	6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
CHR	7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
MAP	8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
MAP	9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
MAP	10	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
MAP	11	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
MAP	12	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
MAP	13	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
MAP	14	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
MAP	15	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

FIGURE 63. Display list command chart

			HSC		HSC		HSC		HSC		HSC		HSC		HSC	HSC	
				VSC	VSC			VSC	VSC			VSC	VSC			VSC	VSC
						LMS	LMS	LMS	LMS					LMS	LMS	LMS	LMS
										DLI	DLI	DLI	DLI	DLI	DLI	DLI	DLI
BLK	1	00								80							
BLK	2	10								90							
BLK	3	20								A0							
BLK	4	30								B0							
BLK	5	40								C0							
BLK	6	50								D0							
BLK	7	60								E0							
BLK	8	70								F0							
JMP		01								81							
JVB		41								C1							
CHR	2	02	12	22	32	42	52	62	72	82	92	A2	B2	C2	D2	E2	F2
CHR	3	03	13	23	33	43	53	63	73	83	93	A3	B3	C3	D3	E3	F3
CHR	4	04	14	24	34	44	54	64	74	84	94	A4	B4	C4	D4	D4	F4
CHR	5	05	15	25	35	45	55	65	75	85	95	A5	B5	C5	D5	E5	F5
CHR	6	06	16	26	36	46	56	66	76	86	96	A6	B6	C6	D6	E6	F6
CHR	7	07	17	27	37	47	57	67	77	87	97	A7	B7	C7	D7	E7	F7
MAP	8	08	18	28	38	48	58	68	78	88	98	A8	B8	C8	D8	E8	F8
MAP	9	09	19	29	39	49	59	69	79	89	99	A9	B9	C9	D9	E9	F9
MAP	A	0A	1A	2A	3A	4A	5A	6A	7A	8A	9A	AA	BA	CA	DA	EA	FA
MAP	B	0B	1B	2B	3B	4B	5B	6B	7B	8B	9B	AB	BB	CB	DB	EB	FB
MAP	C	0C	1C	2C	3C	4C	5C	6C	7C	8C	9C	AC	BC	CC	DC	EC	FC
MAP	D	0D	1D	2D	3D	4D	5D	6D	7D	8D	9D	AD	BD	CD	DD	ED	FD
MAP	E	0E	1E	2E	3E	4E	5E	6E	7E	8E	9E	AE	BE	CE	DE	EE	FE
MAP	F	0F	1F	2F	3F	4F	5F	6F	7F	8F	9F	AF	BF	CF	DF	EF	FF

FIGURE 64. Display list command chart, hex version

Let’s take a look now at the various CHR and MAP modes:

CHR 2 is GRAPHICS 0.

CHR 3 is the same as GRAPHICS 0 except that the characters are 10-scan-lines high instead of 8. This allows you to have lowercase descenders, which means that the tails on “g”, “j”, “p”, “q”, and “y” can drop below the rest of the letters as they should. How do you use this mode? The first step is to redefine the character set. Actually, you only have to change the lowercase letters. What will happen is ANTIC will take the first two bytes of the character description and stick them on the end of the character. It will then make the first two scan lines of the character blank. For non-lowercase characters, it will leave the bytes in order and make the last two scan lines blank. Just in case that doesn't make any sense, let's look at it in pictures (Figure 65).

		In memory:	
	00000000		00001100
	01100110		01111000
	01100110		00000000
	01100110		01100110
	00111100		01100110
	00011000		01100110
	00011000		01100110
	00011000		00111110
		On the screen:	
(0)	## ##		
(1)	## ##		
(2)	## ##		
(3)	####		## ##
(4)	##		## ##
(5)	##		## ##
(6)	##		## ##
(7)			#####
(8)			##
(9)			####

FIGURE 65. Scan lines with lower-case letters

You should note that because each character is now 10-scan-lines high, you can only have 19 rows on the screen (192/10). Make sure of this when you change the display list.

CHR 4 lets you have multi-colored characters. That's right, you can have as many as four different colors in the same character. Let's take a look at how this works.

ANTIC mode four characters are the same size as graphics mode zero characters. The difference, however, is in the size of the pixels that make up the character. The pixels in both modes are one scan line high, but in ANTIC mode four they are as wide as graphics mode seven pixels (one color clock) rather than graphics mode eight. Why? In order to have four colors, a pixel must be represented by at least two bits. That means that the ANTIC mode four pixels have to be twice as wide as those in graphics mode zero, which use one-bit-per-pixel.

Because a character in ANTIC mode four is only four by eight pixels, they aren't of much use for letters and stuff. They are, however, great for graphics. Whatever you end up using them for, How ANTIC interprets a character description byte in this mode is shown in Figure 66.

Now you may be wondering what happens if you try and print a character in inverse video (i.e., an ATASCII value greater than 127). Do all the colors reverse? No, only the pixels with a value of "11" will change; Instead of getting their color from COLOR2 they will get it from COLOR3. This means that you can have all **five** colors at the same time!

To design a character set for this mode, just follow the instructions given for a regular character set, keeping the preceding changes in mind.

BITS	**7 6**	**5 4**	**3 2**	**1 0**
USE	PIXEL 1	PIXEL 2	PIXEL 3	PIXEL 4
PIXEL VALUE	**COLOR REGISTER**			
00	COLOR4 (background)			
01	COLOR0			
10	COLOR1			
11	COLOR2			

FIGURE 66. Character description via ANTIC

CHR 5 is the same as CHR 4 except the characters are now twice as high. Otherwise there's no difference.

CHR 6 is GRAPHICS 1.

CHR 7 is GRAPHICS 2.

MAP 8 is GRAPHICS 3.

MAP 9 is GRAPHICS 4.

MAP 10 is GRAPHICS 5.

MAP 11 is GRAPHICS 6.

MAP 12 is the same as MAP 11 except the pixels are one scan line high instead of two.

MAP 13 is GRAPHICS 7.

MAP 14 is the same as MAP 13 except the pixels are one scan line high instead of two. See DINDEX at location 87 for an example of this mode.

MAP 15 is GRAPHICS 8.

APPENDIX THIRTEEN

SAFE OS VECTORS

Figure 67 is a list of vectors that Atari says will remain the same no matter what changes the OS undergoes. If you want to make sure that your program will run on all the different kinds of Ataris, make sure you don't enter the OS through any other locations.

NAME	ADDR	FUNCTION
EDITRV	58368	Vector table for the screen editor.
SCRENV	58384	Vector table for the display handler.
KEYBDV	58400	Vector table for the keyboard handler.
PRINTV	58416	Vector table for the printer handler.
CASETV	58432	Vector table for the cassette handler
DSKINV	58451	Entry point for the resident disk handler.
CIOV	58454	Entry point for CIO
SIOV	58457	Entry point for SIO.
SETVBV	58460	Routine to set the system timers and VBLANK vectors.
SYSVBV	58463	Entry point for stage one VBLANK.
XITVBV	58466	Exit VBLANK routine.
WARMSV	58484	Entry point for the warm start routine.
COLDSV	58487	Entry point for the cold start routine.

FIGURE 67. Vectors that don't change

You may also use any of the routines in the floating point package.