Atari 2600 Programming for Newbies

By Andrew Davie

# Session 1: Start Here

So, you want to program the Atari 2600 and don't know where to start?

Welcome to the first installment of "000001010 00101000 00000000 1100101"—which at first glance is a rather odd name for a programming tutorial—but on closer examination is appropriate, as it is closely involved with what it's like to program the Atari 2600. The string of 0's and 1's is actually a binary representation of "2600 101".

I'm Andrew Davie, and I've been developing games for various computers and consoles since the late 1970s. Really! What I plan to do with this tutorial is introduce you to the arcane world of programming the '2600, and slowly build up your skill base so that you can start to develop your own games. We'll take this in slow easy stages.

Developing for the Atari 2600 is much simpler today than it was when the machine was a force in the marketplace (back in the 1980s). We have a helpful online community of dedicated programmers, readily available documentation, tools, and sample code—and online forums where we can pose questions and get almost instant feedback and answers. So don't be scared—with a bit of effort, anyone can do this!

It is this online community which makes developing for the machine 'fun'—though I use that in the broadest sense of the word. My 'fun' may be another man's 'torture'. For programming this machine is tricky at best—and not for the feint of heart. But the rewards are great—making this simple hardware do anything at all is quite an achievement—and making it do something new and interesting gives one a warm fuzzy feeling inside.

So, let's get right into it … here's your first installment of "2600 101". We're going to assume that you know how to program \*something\*, but not much more than that. We'll walk through binary arithmetic, hexadecimal, machine architecture, assemblers, graphics, and whatever else gets in our way. And we'll probably divert on tangential issues here and there. But hopefully we'll come out of it with a greater understanding of this little machine, and appreciation for the work of those brilliant programmers who have developed the classics for this system.

## The Basics

A game on the '2600 comes in the form of a cartridge (or 'tape') which is plugged into the console itself. This cartridge consists of a circuit board containing a ROM (or EPROM) which is basically just a silicon chip containing a program and graphics for displaying the game on your TV set. This program (and graphics) are really just a lot of numbers stored on the ROM which are interpreted by the CPU (the processor) inside your '2600 just like a program on any other computer. What makes the '2600 special is … nothing. It's a computer, just like any other!

A computer typically consists of a CPU, memory, and some input/output (I/O) systems. The '2600 has a CPU (a 6507), memory (RAM for the program's calculations, ROM to hold the program and graphics), and I/O systems (joystick and paddles for input, and output to your TV).

### The CPU

The CPU of the '2600 is a variant of a processor used in computers such as the Apple II, the Nintendo NES, the Super Nintendo, and Atari home computers (and others). It's used in all these machines because it is cheap to manufacture, it's simple to program, but also effective—the famous '6502'. In this course we will learn how to program the 6502 microprocessor … but don't panic, we'll take that in easy stages (and besides, it's not as hard as it looks).

The '2600 actually uses a 6507 microprocessor—but this is really just a 6502 dressed in sheep's clothing. The 6507 is able to address less memory than the 6502 but is in all other respects the same. I refer to the '2600 CPU as a 6502 purely as a matter of convenience.

### Memory

Memory is severely restricted on the '2600. When the machine was developed, memory (both ROM and RAM) were very expensive, so we don't have much of either. In fact, there's only 128 BYTES of RAM (and we can't even use all of that!)—and typically (depending on the capabilities of the cartridge we're going to be using for our final game) only about 4K of ROM. So, then, here's our first introduction to the 'limitations' of the machine. We may all have great ideas for '2600 games, but we must keep in mind the limited amount of RAM and ROM!

[If you'd like to create and sell games with a lot more ROM and RAM, check out the Melody boards at AtariAge. And be sure to pick up a Harmony Cartridge for testing your games on a real Atari 2600.]

### Input/Output

Input to the '2600 is through interaction by the users with joystick and paddle controllers, and various switches and buttons on the console itself. There are also additional control devices such as keypads—but we won't delve much into those. Output is invariably through a television picture with sound (the game that we see on our TV).

So, there's not really much to it so far—we have a microprocessor running a program from ROM, using RAM, as required, for the storage of data—and the output of our program being displayed on a TV set. What could be simpler?

## The Development Process

Developing a game for the '2600 is an iterative process involving editing source code, assembling the code, and testing the resulting binary (usually with an emulator). Our first step is to gather together the tools necessary to perform these tasks.

### Source Code

'Source code' is simply one or more text files (created by the programmer and/or tools) containing a list of instructions (and 'encoded' graphics) which make up a game. These data are converted by the assembler into a binary which is the actual data placed on a ROM in a cartridge, and is run by the '2600 itself.

### Text Editor

To edit your source code, you need a text editor—and here the choice is entirely up to you. I use Microsoft Developer Studio myself, as I like its features—but any text editor is fine. Packages integrating the development process (edit/assemble/test) into your text editor are available, and this integration makes the process much quicker and easier (for example, Developer-Studio integration allows a double-click on an error line reported by the assembler, and the editor will position you on the very line in the source code causing the error).

### Assembler

To convert your source code into a binary form, we use an 'assembler'. An assembler is a program which converts assembly language into binary format (and in particular, since the '2600 uses a 6502-variant processor, we need an assembler that knows how to convert 6502 assembly code into binary). Pretty much all '2600 development these days is done using the excellent cross-platformAvailable for multiple machines such as Mac, Linux, Windows, etc. assembler 'DASM' which was written by Matt Dillon in about 1988.

DASM is available online. It would be a good idea to go there now and get a copy of DASM, and the associated support-files for '2600 development. In this course, we will be using DASM exclusively. We'll learn how to setup and use DASM shortly. [DASM is also included with batari Basic, so you could get it there if you don't trust other sources.]

### Emulator

Development of a game in the 1980s consisted of creating a binary imageWrite source code, assemble into binary. and then physically 'burning' the binary onto an EPROM, putting that EPROM onto a cartridge and plugging it into a '2600. This was an inherently slow process (trust me, I did this for NES development!) and it sometimes took 15 minutes just to see a change!

Nowadays, we are able to see changes to code almost immediately because of the availability of good emulators. An emulator is a program which pretends to be another machine/program. For example, a '2600 emulator is able to 'run' binary ROM images and display the results just as if you'd actually plugged a cartridge containing a ROM with that binary into an actual '2600 console. Today's '2600 emulators are very good indeed.

So, instead of actually burning a ROM, we're just going to pretend we've burned one—and look at the results by running this pretend-ROM on an emulator. And if there's a problem, we go back and edit our source code, assemble it to a binary, and run the binary on the emulator again. That's our iterative development process in action.

There are quite a few '2600 emulators available, but two of note are:

* Stella
* z26

Stella is your best choice if you're programming on non-Windows platform. I use Z26 for Windows development, as it is quite fast and appears to be very accurate. Either of these emulators is fine, and it's handy to be able to cross-check results on either.

We'll learn how to use these emulators later—but right now let's continue with the gathering of things we need…

Now that we have an editor, an assembler, and an emulator—the next important things are documentation and sources for information. There are many places on the 'net where you can find information for programming '2600, but perhaps the most important are:

* The Stella List
* AtariAge

## Documentation

And finally, documentation. A copy of the technical specifications of the '2600 hardware (the Stella Programmer's Guide) is essential…

* Stella Programmer's Guide

## Summary

OK, that's all we need. Here's a summary of what you should have…

* Text editor of your choice.
* DASM assembler and '2600 support files.
* Emulator (Z26 or Stella)
* Stella Programmer's Guide
* Bookmarks to AtariAge and the #Stella mailing list.

That's it for this session. Have a read of the Stella Programmer's Guide (don't worry about understanding it yet), and try installing your emulator (and play a few games for 'research' purposes). Next time we will make sure that our development environment is setup correctly, and start to discuss the principles of programming a '2600 game.

P.S. I can't promise to complete this 'course'—but hopefully what I do write will be interesting and helpful.

# Session 2: Television Display Basics

Hopefully you've been through the first part and have your editor, assembler, emulator and documentation ready to go. What we're going to look at now is a basic overview of how a television works, and why this is absolutely necessary prerequisite knowledge for the '2600 programmer. We're not going to cover a lot of '2600 specific stuff this time, but this is most definitely stuff you NEED TO KNOW!

Television has been around longer than you probably realize. Early mechanical television pictures were successfully broadcast in the '20s and '30s (yes, really!—see tvdawn.com). The mechanical 'scanning' technology utilized in these early television systems are no doubt the predecessors to the 'scanning' employed in our modern televisions.

A television doesn't display a continuous moving image. In fact, television displays static (non-moving) images in rapid succession—changing between images so quickly that the human eye perceives any movement as continuous. And even those static images aren't what they seem—they are really composed of lots of separate lines, each drawn one after the other by your TV, in rapid succession. So quick, in fact, that hundreds of them are drawn every image, and many images are drawn every second. In fact, the actual numbers are very important, so we'll have a look at those right now.

## NTSC, PAL, and SECAM

The Atari 2600 console was released in many different countries around the world. Not all of these countries use the same television 'system'—in fact there are three variations of TV systems (and there are three totally different variations of Atari 2600 hardware to support these systems). These systems are called NTSC, PAL, and SECAM. NTSC is used for the USA and Japan, PAL for many European countries, and Australia, and SECAM is used in France, some ex-French colonies (for example, Vietnam), and Russia. SECAM is very similar to PAL (625/50Hz), but I won't spend much time talking about it, as Atari SECAM units are incredibly rare, and little if any development is done for that format anyway. Interestingly, the differences in requirements for displaying a valid TV image for these systems leads to the incompatibility between cartridges made for NTSC, PAL and SECAM Atari units. We'll understand why, shortly!

### Images Per Second (Frequency)

A television signal contains either 60 images per second (on NTSC systems) or 50 images per second (on PAL systems). This is closely tied to the frequency of mains AC power in the countries which use these systems—and this is probably for historical reasons. In any case, it's important to understand that there are differences. Furthermore, NTSC images are 525 scanlines deep, and PAL images are 625 scanlines deep. From this, it follows that PAL images have more detail—but are displayed less frequently—or alternatively, NTSC images have less detail but are displayed more often. In practice, TV looks pretty much the same in both systems.

But from the '2600 point of view, the difference in frequency (50Hz vs. 60Hz) and resolution (625 scanlines vs. 525 scanlines) is important—very important—because it is the PROGRAMMER who has to control the data going to the TV. It is not done by the '2600 (!!)—the '2600 only generates a signal for a single scanline.

This is completely at odds with how all other consoles work, and what makes programming the '2600 so much 'fun'. Not only does the programmer have to worry about game mechanics—but he or she also has to worry about what the TV is doing (for example, what scanline it is drawing, and when it needs to start a new image, etc.).

Let's have a look at how a single image is drawn by a TV…

### Scanline

A television is a pretty amazing piece of 1930's technology. It forms the images we see by shining an electron beam (or 3, for color TVs) onto a phosphor coating on the front of the picture tube. When the beam strikes the phosphor, the phosphor starts to glow—and that glow slowly decreases in brightness until the phosphor is next hit by the electron beam. The TV 'sweeps' the electron beam across the screen to form 'scanlines'—at the same time as it sweeps, adjusting the intensity of the beam, so the phosphor it strikes glow brightly or dimly. When the beam gets to the end of a scanline, it is turned off, and the deflection circuitry (which controls the beam) is adjusted so that the beam will next start a little bit down, and at the start (far left-hand-side) of the next scanline. And it will then turn on, and sweep left-to-right to draw the next scanline. When the last scanline is drawn, the electron beam is turned off, and the deflection circuitry is reset so that the beam's position will next be at the top left of the TV screen—ready to draw the first scanline of the next frame.

This 'turning-off' and repositioning process—at the end of a scanline, and at the end of an image—is not instantaneous—it takes a certain amount of time for the electronics to do this repositioning, and we'll understand this when we come to talk about the horizontal blank (when the beam is resetting to the left of the next scanline) and the vertical blank (when the beam is resetting to the top left scanline on the screen). I'll leave that for a later session, but when we do come to it, you'll understand what the TV is doing at these points.

A fairly complex—but nonetheless simple-to-understand analog signal controls the sweeping of the electron beam across the face of the TV. First it tells the TV to do the repositioning to the start of the top left line of the screen, then it includes color and intensity information for the electron beam as it sweeps across that line, then it tells the TV to reposition to the start of the next scanline, etc., right down to the last scanline on the screen. Then it starts again with another reposition to the start… That's pretty much all we need to know about how that works.

The Atari 2600 sends the TV the "color and intensity information for the electron beam as it sweeps across that line", and a signal for the start of each new line. The '2600 programmer needs to feed the TV the signal to start the image frame.

#### Interlacing

A little side-track, here. Although I stated that the vertical resolution of a TV image is 625 lines (PAL) and 525 lines (NTSC), television employs another 'trick' called interlacing. Interlacing involves building up an image out of two separate 'frames'—each frame being either the odd scanlines, or the even scanlines of that image. Each frame is displayed every 1/30th of a second (30Hz) for NTSC, or every 1/25th of a second (25Hz) for PAL. By offsetting the vertical position of the start of the first scanline by half a scanline, and due to the persistence of the phosphor coating on the TV, the eye/brain combines these frames displaying alternate lines into a single image of greater vertical resolution than each frame. It's tricky and messy, but a glorious 'hack' solution to the problem of lack of bandwidth in a TV signal.

The upshot of this is that a single FRAME of a TV image is actually only half of the vertical resolution of the image. Thus, a NTSC frame is 525/2 = 262.5 lines deep, and a PAL frame is 625/2 = 312.5 lines deep. The extra .5 of a line is used to indicate to the TV if a frame is the first (even lines) or second (odd lines) of an image. An aside: about a year ago, the #stella community discussed this very aspect of TV images, and if it would be possible for the Atari to exploit this to generate a fully interlaced TV frame—and, in fact, it is possible. So some 25 years after the machine was first released, some clever programmers discovered how to double the resolution of the graphics.

Back to basics, though. We just worked out that a single frame on a TV is 262.5 (NTSC) and 312.5 (PAL) lines deep. And that extra .5 scanline was used to tell the TV if the frame was odd or even. So the actual depth of a single frame is 262 (NTSC) and 312 (PAL) lines. Now, if TV's aren't told that a frame is odd, they don't offset the first scanline by half a scanline's depth—and so, scanlines on successive frames are exactly aligned. We have a non-interlaced image, displayed at 60Hz (NTSC) or 50Hz (PAL). And this is the 'standard' format of an Atari 2600 frame sent to a TV.

In summary, an Atari 2600 frame consists of 262 scanlines (NTSC) or 312 scanlines (PAL), sent at 60Hz (NTSC) or 50Hz (PAL) frequency. It is the job of the '2600 programmer to make sure that the correct number of scanlines are sent to the TV at the right time, with the right graphics data, and appropriate control signals to indicate the end of the frame are also included.

### Color Encoding

One other aspect of the difference between TV standards—and a consequence of the incremental development of television technology (first we had black and white, then color was added—but our black and white TVs could still display a color TV signal—in black and white)—is that color information is encoded in different places in the signal for NTSC and PAL (and SECAM) systems. So, even though the programmer is fully-responsible for controlling the number of scanlines per frame, and the frequency at which frames are generated, it is the Atari itself which encodes the color information into the TV signal.

This is the fundamental reason why there are NTSC, PAL, and SECAM Atari systems—the encoding of the color information for the TV signal! We get some interesting combinations of Atari and games, for example…

If we plug a NTSC cartridge into a PAL '2600, then we know that the NTSC game is generating frames which are 262 lines deep, at 60Hz. But a PAL TV expects frames 312 lines deep, at 50Hz. So the image is only 262/312 of the correct depth, and also images are arriving 60/50 times faster than expected. If we were viewing on a NTSC TV, then the PAL console would be placing the color information for the TV signal in a completely different place than the TV is expecting—so we would see our game in black and white.

There are several combinations you can play with—but the essence is that if you use a different '2600 variant than TV, you will only get black and white (for example, NTSC '2600 with PAL TV or PAL '2600 with NTSC TV) as the color information is not in at the correct frequency band of the signal. And if you plug in a different cartridge than TV (NTSC cart with PAL TV or vice-versa) then what you see depends on the television's capability to synchronize with the signal being generated—as it is not only the incorrect frequency, but also the incorrect number of scanlines.

## Summary

All of this may sound complicated—but really all we need to do is create a 'kernel' (which is the name for your section of an Atari 2600 program which generates the TV frame) which does the drawing correctly—and once that's working, we don't really need to worry too much about the TV—we can abstract that out and just think about what we want to draw.

Well, I lie, but don't want to scare you off TOO early. Wink

Next time, let's have a look how the processor interacts with hardware, I/O and memory.

# Sessions 3 & 6: The TIA and the 6502

Let's spend this session having a look at how some of the hardware generates a scanline for the TV. Remember in session 2, we had a good look at how a TV works, and in particular how a TV frame is composed of 262 scanlines (NTSC) or 312 scanlines (PAL). It's the programmer's job to control how many scanlines are sent to the TV, but it is the '2600 which builds the actual signal comprising the color and intensity information for any scanline. This color and intensity information is derived from the internal 'state' of the TIA (Television Interface Adaptor) chip inside the '2600. The TIA is responsible for creating the signal for a single scanline for the TV.

## The TIA

The TIA 'draws' the pixels on the screen 'on-the-fly'. Each pixel is one 'clock' of the TIA's processing time, and there are exactly 228 color clocks of TIA time on each scanline. But a scanline consists of not only the time it takes to scan the electron beam across the picture tube, but also the time it takes for the beam to return to the start of the next line (the horizontal blank, or retrace). Of the 228 color clocks, 160 are used to draw the pixels on the screen (giving us our maximum horizontal resolution of 160 pixels per line), and 68 are consumed during the retrace period.

### 6502 Clock

The 6502 clock is derived from the TIA clock through a divide-by-three. That is, for every single clock of 6502 time, three clocks of TIA time have passed. Therefore, there are \*exactly\* 228/3 = 76 cycles of 6502 time per scanline. The 6502 and TIA perform a complex 'in-step' dance—one cycle of 6502, three cycles of TIA. A side-note: 76 cycles per line x 262 lines per frame x 60 frames per second = the number of 6502 cycles per second for NTSC (roughly equals 1.19MHz).

So, as our 6502 program is executing its instructions, the TIA is also sending data for each scanline. Every cycle of 6502 time we know that the TIA has sent 3 color clocks of information to the TV. If the TIA was in the first 68 color clocks of the scanline, then it was in the horizontal retrace period. If it was in color clock 68-227, then it was drawing pixels on the visible scanline. And so we go, the 6502 program doing its stuff and at the very same time the TIA doing its stuff.

The magic happens when you start changing the 'state' of the TIA, because those changes are reflected immediately in the TIA output to the TV! Since the 6502 is 'locked' to the TIA through their shared timing origin, it is possible for the programmer to know exactly what pixel on a scanline the TIA is currently drawing. And knowing where the TIA 'is at' allows us to change what it is drawing at particular positions on the scanline. We don't have much scope for change, but we do have some. And it is this ability that master '2600 programmers use to achieve all those amazing effects.

Naturally, to achieve this sort of precision timing, programmers have to know exactly how long the 6502 takes to do each instruction. For example, a load/store combination takes a minimum of 5 cycles of 6502 time. How many onscreen pixels is that? Remember, 3 color clocks per 6502 cycle, so that's 3 x 5 = 15 pixels. Essentially, if one were using the quickest possible load/store combinations to change the color of, say, the background, then the absolute quickest this could be done would be every 15 pixels (just on 11 times per scanline).

### TV Timing

Here's an updated image of the TV timing, taken from the Stella Programming Guide. Some of the numbers should make sense, now. The ones that don't … we'll cover those soon.

Have a good look at this image, and try and understand what it's showing. Your understanding of this will greatly assist your '2600 programming efforts, especially when it comes to designing your kernel.

A screen shot of a video game

Description automatically generated

## Summary

Don't despair! It is not necessary for you to learn how to count 6502 cycles at this stage. Those sort of tricks are for more advanced '2600 programming—and the original design of the TIA hardware made this unnecessary. It's only when you need to push the hardware (TIA) beyond its original design, that you will come to appreciate the benefit inherent in the way that the 6502 and TIA are intricately tied together.

Next session we'll have a closer look at the TIA and how it determines what color to use for each pixel of the scanline it is drawing. In particular, we'll start to look at background, playfield, sprite, missile and ball graphics.

# Session 4: The TIA

Last session we were introduced to the link between the 6502 and the TIA. Specifically, how every cycle of 6502 time corresponds to three color clocks of TIA time.

## The TIA

The TIA determines the color of each pixel based on its current 'state', which contains information about the color, position, size and shape of objects such as background, playfield, sprites (2), missiles (2) and ball. As soon as the TIA completes a scanline (228 cycles, consisting of 160 color clocks of pixels, and 68 color clocks of horizontal blank), it begins drawing the next scanline. Unless there is some change to the TIA's internal 'state' during a scanline, then each scanline will be absolutely identical.

Consequently, the absolute simplest way to 'draw' 262 lines for a NTSC frame is to just WAIT for 262 (lines) x 76 (cycles per line) 6502 cycles. After that time, the TIA will have sent 262 identical lines to the TV. There are other things that we'd need to do to add appropriate control signals to the frame, so that the TV would correctly sync to the frame—but the essential point here is that we can leave the TIA alone and let it do its stuff. Without our intervention, once the TIA is started it will keep sending scanlines (all the same!) to the TV. And all we have to do to draw n scanlines is wait n x 76 cycles.

It's time to have a little introduction to the 6502.

### Binary Numbers

The CPU of the '2600, the 6502, is an 8-bit processor. Basically this means that it is designed to work with numbers 8-binary-bits at a time. An 8-bit binary number has 8 0's or 1's in it, and can represent a decimal number from 0 to 255. Here's a quick low-down on binary…

In our decimal system, each digit 'position' has an intrinsic value. The units position (far right) has a value of 1, the tens position has a value of 10, the hundreds position has a value of one hundred, the thousands position has a value of 1000, etc. This seems silly and obvious—but it's also the same as saying the units position has a value of 10^0 (where ^ means to the power of), the tens position has a value of 10^1, the hundreds position has a value of 10^2, etc. In fact, it's clear to see that position number 'n' (counting right to left, from n=0 as the right-most digit) has a value of 10^n.

That's true of ANY number system, where the 10 is replaced by the 'base'. For example, hexadecimal is just like decimal, except instead of counting 10 digits (0 to 9) we count 16 digits (0 to 15, commonly written 0 1 2 3 4 5 6 7 8 9 A B C D E F—thus 'F' is actually a hex digit with decimal value 15—which again, is 1 x 10^1 + 5 x 10^0 ). So in hexadecimal (or hex, for short), the digit positions are 16^n. There's no difference between hex, decimal, binary, etc., in terms of the interpretation of a number in that number system. Consider the binary number 01100101—this is (reading right to left) … 1 x 2^0 + 0 x 2^1 + 1 x 2^2 + 0 x 2^3 + 0 x 2^4 + 1x2^5 + 1x2^6 + 1x2^7. In decimal, the value is 101. So, %01100101 = 101 where the % represents a binary number. Hexadecimal numbers are prefixed with a $.We'll get used to using binary, decimal and hex interchangeably—after all they are just different ways of writing the same thing. When I'm talking about numbers in various bases, I'll include the appropriate prefix when not base-10.

So now it should be easy to understand WHY an 8-bit binary number can represent decimal values from 0 to 255—the largest binary number with 8 bits would be %11111111—which is 1 x 2^7 + 1 x 2^6 + … + 1 x 2^0.

The 6502 is able to shift 8-bit numbers to and from various locations in memory (referred to as addresses)—each memory location is \*UNIQUELY\* identified by a memory address, which is just like your house street address, or your post-box number. The processor is able to access memory locations and retrieve 8-bit values from, or store 8-bit values to those locations.

### Registers

The processor itself has just three 'registers'. These are internal memory/storage locations. These three registers (named 'A', 'X', and 'Y') are used for manipulating the 8-bit values retrieved from memory locations and for performing whatever calculations are necessary to make your program do its thing.

What can you do with just three registers? Not much … but a hell of a lot of not much adds up to something! Just like with the TV frame generation, a lot of work is left for the programmer. The 6502 cannot multiply or divide. It can only increment, decrement, add and subtract, and it can only work with 8-bit numbers! It can load data from one memory location, do one of those operations on it (if required) and store the data back to memory (possibly in another location). And out of that capability comes all the games we've ever seen on the '2600. Amazing, innit?

At this stage it is probably a good idea for you to start looking for some books on 6502 programming—because that's the ONLY option when programming '2600. Due to the severe time, RAM and ROM constraints, every cycle is precious, every bit is sacred. Only the human mind is currently capable of writing programs as efficiently as required for '2600 development.

That was a bit of a diversion—let's get back to the TIA and how the TIA and 6502 can be used together to draw exactly 262 lines on the TV. Our first task is simply to 'wait' for 76 cycles, times 262 lines.

### NOP

The simplest way to just 'wait' on the 6502 is just to execute a 'nop' instruction. 'nop' stands for no-operation, and it takes exactly two cycles to execute. So if we had 38 'nop's one after the other, the 6502 would finish executing the last one exactly 76 cycles after it started the first. And assuming the first 'nop' started at the beginning of the scanline, then the TIA (which is doing its magic at the same time) would have just finished the last color clock of the scanline at the same time as the last nop finished. In other words, the very next scanline would then start as our 6502 was about to execute the instruction after the last nop, and the TIA was just about to start the horizontal retrace period (which, as we have learned, is 68 color clocks long).

How do we tell the 6502 to execute a 'nop'? Simply typing nop on a line by itself (with at least one leading space) in the source code is all we have to do. The assembler will convert this mnemonic into the actual binary value of the nop instruction. For example…

; sample code

    NOP

    nop

; end of sample code

The above code shows two nop instructions—the assembler is case-insensitive. Comments are preceded by semicolons, and occupy the rest of a line after the ; Opcodes (instructions) are mnemonics—typically 3 letters—and must not start at the beginning of a line! We can have only one opcode on each line. An assembler would convert the above code into a binary file containing two bytes—both $EA (remember, a $ prefix indicates a hexadecimal number) = 234 decimal. When the 6502 retrieves an opcode of $EA, it simply pauses for 2 cycles, and then executes the next instruction. The code sequence above would pause the processor for 4 cycles (which is 12 pixels of TIA time, right?!)

But there are better ways to wait 76 cycles! After all, 38 'nop's would cost us 38 bytes of precious ROM—and if we had to do that 262 times (without looping), that would be 9432 bytes—more than double the space we have for our ENTIRE game!

### WSYNC

The TIA is so closely tied to the 6502 that it has the ability to stop and start the 6502 at will. Funnily enough, at the 6502's will! More correctly, the 6502 has the ability to tell the TIA to stop it (the 6502), and since the TIA automatically re-starts the 6502 at the beginning of every scanline, the very next thing the 6502 knows after telling the TIA to stop the CPU is that the TIA is at the beginning of the very next scanline. In fact, this is the way to synchronize the TIA and 6502 if you're unsure where you're at—simply halt the CPU through the TIA, and next thing you know you're synchronized. It's like a time-warp, or a frozen sleep—you're simply not aware of time passing—you say 'halt' and then continue on as if no halt has happened. It has, but the 6502 doesn't know it.

This CPU-halt is achieved by writing any value to a TIA 'register' called WSYNC. Before we get into reading and writing values to and from 'registers' and 'memory', and what that all means, we'll need to have a look at the memory architecture of the '2600—and how the 6502 interacts with memory, including RAM and ROM.

## Summary

We'll start to explore the memory map (architecture) and the 6502's interaction with memory and hardware, in our next installment.

# Session 5: Memory Architecture

Let's have a look at the memory architecture of the '2600, and how the 6502 communicates with the TIA and other parts of the '2600 hardware.

## Memory Mapping

The 6502 communicates with the TIA by writing, and sometimes reading values to/from TIA 'registers'. These registers are 'mapped' to certain fixed addresses in the 6502's addressing range.

In its simplest form, the 6502 is able to address 65536 (2^16) bytes of memory, each with a unique address. Each 16-bit address ultimately directly controls the 'wires' on a 16-bit buspathway to memory, selecting the appropriate byte of memory to read/write. However, the '2600 CPU, the 6507, is only able to directly access 2^13 bytes (8192 bytes) of memory. That is, only 13 of the 16 address lines are actually connected to physical memory.

This is our first introduction to 'memory mapping' and mirroring. Given that the 6507 can only access addresses using the low 13 bits of an address, what happens if bit 14, 15, or 16 of an address are set? Where does the 6507 go to look for its data? In fact, bits 14,15, and 16 are totally ignored—only the low 13 bits are used to identify the address of the byte to read/write. Consider the valid addresses which can be formed with just 13 bits of data…

from %0000000000000 to %1111111111111

= from $0000 to $1FFF

### Zero is Zero

Note: $0000 is the same as 0 is the same as %000 is the same as %0000000000. 0 is 0. In the same vein, any number with leading zeros is the same as that number without zeros. I often see people writing $02 when they could just write $2, or better yet … 2. Your assembler doesn't care how numbers are written. It's the value of numbers that matter. So use the most readable form of numbers, where it makes sense. Remember, 0 is 0000 is %0 is $000

### Memory Footprint

So we've just written down the minimum and maximum addresses that can be formed with 13 bits. This gives us our memory 'footprint'—the absolute extremes of memory which can be accessed by the 6507 through a 13-bit address.

### Reads and/or Writes

This next idea is important, so make sure you understand! All communication between the CPU and hardware (be it ROM, RAM, I/O, the TIA, or other) is through reads and/or writes to memory locations. Read that again.

The consequences of this are that some of that memory range (between $0 and $1FFF) must contain our RAM, some must contain our ROM (program), and some must presumably allow us to communicate with the TIA and whatever other communication/control systems the machine has. And that's exactly how it works.

### RAM

We have just 128 bytes of RAM on the '2600. That RAM 'lives' at addresses $80 - $FF. It's always there, so any write to location $80 (128 decimal) will actually be to the first byte of RAM. Likewise, any read from those locations is actually reading from RAM.

So we've just learned that the 6507 addresses memory using 13 bits to uniquely identify the memory location, and that some areas of that memory 'range' are devoted to different uses. The area from $80 to $FF is our 128 bytes of RAM!

Don't worry too much about understanding this yet, but TIA registers are mapped in the memory addresses 0 to $7F, RIOT (a bit of '2600 hardware we'll look at later) from $280 - $2FF (roughly), and our program is mapped into address range $1000 to $1FFF (a 4K size).

Note: 1K = 1024 bytes = $400 bytes = %10000000000 bytes.

### The TIA

In essence, then, to change the state of the TIA we just have to write values to TIA 'registers' which look to the 6507 just like any other memory location and which 'live' in addresses 0 to $7F. To the 6502 (and I'll revert to that name now we've emphasized that the 6507 only has 13 address lines as opposed to the 6502's 16 and all other things are equal) a read or write of a TIA register is just the same as a read or write to any other area of memory. The difference is, the TIA is 'watching' those locations, and when you write to that memory, you're really changing the TIA 'registers'—and potentially changing what it draws on a scanline.

## Summary

So now we know how to communicate with the TIA, and where it 'lives' in our memory footprint. And we know how to communicate with RAM, and where it 'lives'. Even our program in ROM is really just another area in our memory 'map'—the program that runs from a cartridge is accessed by the 6502 just by reading memory locations. In effect, the cartridge 'plugs-in' to the 6502 memory map. Let's have a quick look at what we know so far about memory…

|  |  |
| --- | --- |
| Address Range | Function |
| $0000 - $007F | TIA registers |
| $0080 - $00FF | RAM |
| $0200 - $02FF | RIOT registers |
| $1000 - $1FFF | ROM |

We'll keep it simple for now—though you may be wondering what 'lives' in the gaps in that map, between the bits we know about. The short answer is 'not much'—so let's not worry about those areas for now. Just remember that when we're accessing TIA registers, we're really accessing memory from 0 to $7F, and when we access RAM, we're accessing memory from $80 to $FF, etc.

Now that we understand HOW the 6502 communicates with the TIA, one of our next steps will be to start to examine the registers of the TIA and what happens when you modify them. It won't be long now before we start to understand how it all works. Stay tuned.

I might give up writing "next time we'll talk about…" because I seem to end up covering something completely different.

# Session 7: The TV and our Kernel