2600 Programming for newbies

By: Andrew Davie

Hosted by

AtariAge

http://www.atariage.com/forums/viewforum.php?f=31

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, and I encourage you to ask questions - this will help me pace the tutorial and introduce subjects of interest.

Developing for the Atari 2600 is much simpler today than it was when the machine was a force in the marketplace (i.e.: 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 have 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!

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) - i.e.: 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' 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.

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).

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-platform (i.e.: versions are available for multiple machines such as Mac, Linux, Windows, etc) assembler 'DASM' which was written by Matt Dillon in about 1988.

DASM is now supported by yours-truly, and is available at "http://www.atari2600.org/dasm" - it would be a good idea now to go there 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.

Development of a game in the '80s consisted of creating a binary image (i.e.: write 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

Z26 - available at http://www.whimsey.com/z26/
Stella - available at http://sourceforge.net/projects/stella/

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 - at http://www.biglist.com/lists/stella/
AtariAge - at http://www.atariage.com/

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

Stella Programmer's Guide
- text version at http://stella.sourceforge.net/download/stella.txt
- PDF version at http://www.atarihq.com/danb/files/stella.pdf

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.

PS: 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 pre-requisite 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 http://www.tvdawn.com/index.htm). 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.

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 (e.g.: 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!

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 she also has to worry about what the TV is doing (i.e.: what scanline it is drawing, and when it needs to start a new image, etc., etc).

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

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 colour 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 colour 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 "colour 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.

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 (i.e.: at 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 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.

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 colour was added - but our black and white TVs could still display a colour TV signal - in black and white) - is that colour 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 colour information into the TV signal.

This is the fundamental reason why there are NTSC, PAL, and SECAM Atari systems - the encoding of the colour 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 colour 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 (e.g.: NTSC '2600 with PAL TV or PAL '2600 with NTSC TV) as the colour information is not in at the correct frequency band of the signal. And if you plug in a different cartridge than TV (e.g.: 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.

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

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

The three major differences between NTSC & PAL/SECAM are:
1. Frame rate (60fps vs 50fps). This impacts any calculations which are based on the number of frames, typically movement.
2. Number of lines per frame (262 vs 312), important since the game is responsible for triggering VSYNC.
3. Number of visible lines (192 vs 228), which can change all kinds of things like sprite sizes, size

of the playing field, vertical motion rate, and more.

Some homebrew programmers use the TV switch to change between NTSC and PAL, while others create NTSC & PAL specific versions.

(SECAM is the same as PAL (50/312/228) except the TV switch is hardwired to B&W and there are only 8 colors.)

PS The 2600 has 160 pixels of horizontal resolution.

Please read the tutorial carefully. It explains what happens when you play a game made for one TV system, on another TV system.

If a company *converts* their ROM from one format to the other, then they also have the option of changing the sizes of things. Typically they wouldn't do that, so your statement that a PAL game converted to NTSC will generally display bigger sprites is somewhat correct. But in doing that conversion, you will also have to remove 50 scanlines from the screen, so you may very well have to reduce the size of your sprites to fit things in after all!

It's all a balancing act.

Worrying about the sizes of things between PAL and NTSC is pretty much the least of your worries. Just pick a platform (NTSC is my recommendation) and program your game for it. You can tackle the system conversion at a later date. And nobody is going to be running your NTSC game on a PAL console. And if they do it will be in black and white. Read the tutorial!

Session 3: 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 colour and intensity information for any scanline. This colour and intensity information is derived from the internal 'state' of the TIA (Television Interface Adapter) chip inside the '2600. The TIA is responsible for creating the signal for a single scanline for the TV.

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 colour 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 colour 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.

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 (= 1.19MHz, roughly).

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 colour clocks of information to the TV. If the TIA was in the first 160 colour clocks of the scanline, then it was drawing pixels. If it was in colour clock 160-227, then it was handling horizontal blank. 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 where on a scanline the TIA is currently drawing (i.e.: what pixel). 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 colour 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 colour of, say, the background, then the absolute quickest this could be done would be every 15 pixels (i.e.: just on 11 times per scanline).

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 colour 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 colour clocks of TIA time.

The TIA determines the colour of each pixel based on its current 'state', which contains information about the colour, 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 colour clocks of pixels, and 68 colour 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 synch 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.

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.

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, isn't it?

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.

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 colour 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 draw the first pixel.

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...

Code:

```
; 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!

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.

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.

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 bus (=pathway) 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

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

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.

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.

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.

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.

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.

In Andrew's sample kernel he uses ORG $F000 to set the start of the code. He then uses ORG $FFFA to set the start of the 3 interrupt vectors. Each vector is 2 bytes, for a total of 6 bytes. $FFFA + 6 = $10000. $10000 - $F000 = $1000 = 4096 = 4K.

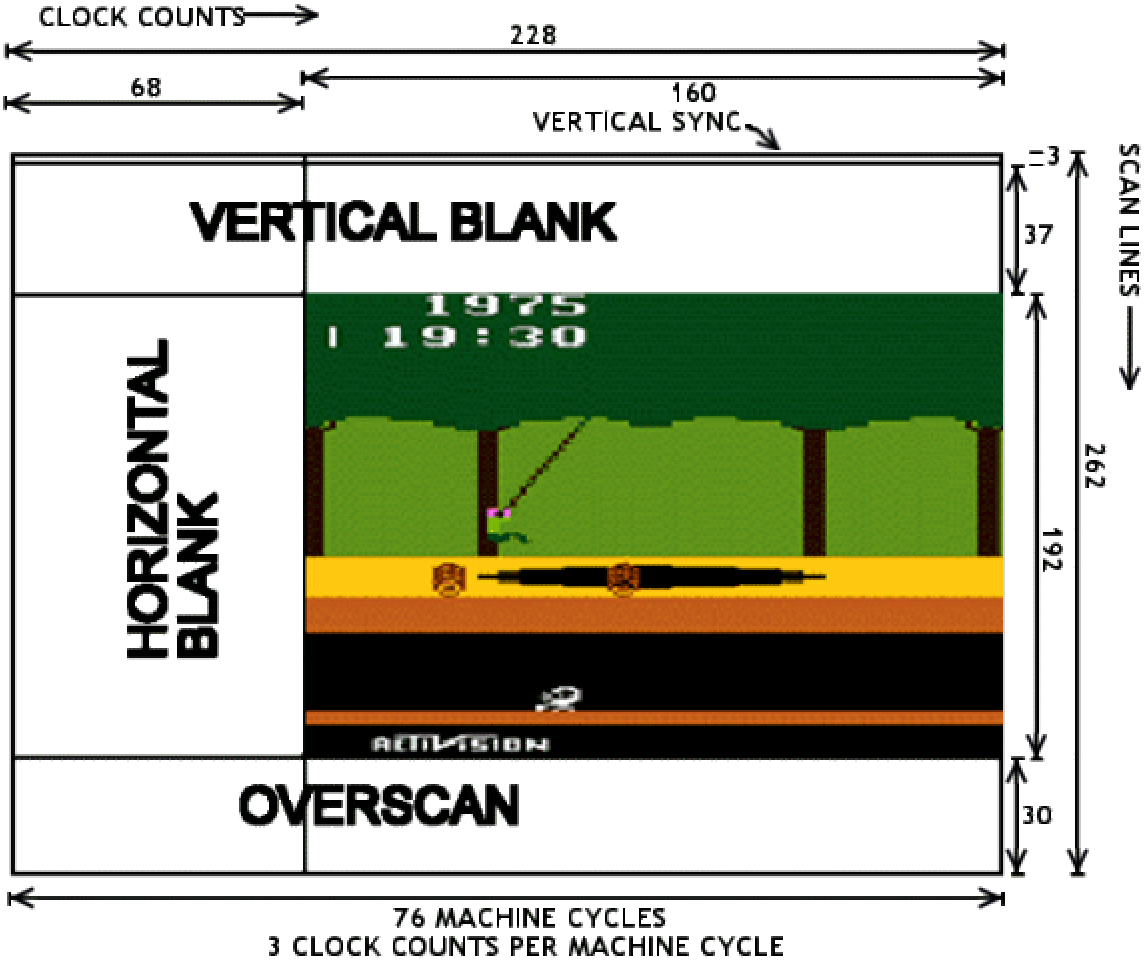
Now that I have that clarified I can answer your question. If you recall, the 6507 used in the 2600 only has 13 address lines. Thus from the 6507's perspective $F000 is the same as $1000 because the 13 least significant bits are the same. And when DASM generates the code (with the -f3 option) it starts the binary file at the first ORG. So DASM only generates a 4K file.

It really comes down to (for 4K games at least) style. Though if you want to use a 6502 emulator (instead of a 2600 emulator) you will need to put the interrupt vectors down at $FFFA (instead of $1FFA) so the emulator can find them, since the 6502 emulator will track all 16 bits of the address.

Session 6: TV Timing Diagram

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.



Since general division & multiplication take so many cycles (often with a wide variation between cases) 2600 programmers try to avoid them if at all possible. Where one of the operands is a constant a dedicated routine can often be created (such as the div/mod 15 routine sometimes used for sprite positioning) or a table used. But even then, it is a good idea to try and figure out a way to do without.

BCD (binary coded decimal, where each byte stores 2 decimal digits, 00-99) sometimes is used for scores, etc where the display is in decimal. But often it's easier to use one byte (or even a two byte pointer) per decimal digit (so you don't have to try and extract each nibble).

Session 7: The TV and our Kernel

Time to complete our understanding of what constitutes a TV frame - exactly what has to be sent to the TV to make it display a picture correctly.

Revisit the diagram posted earlier, with the timing information and the Pitfall! image inside. Your understanding of the numbers across the top should be good, but just to briefly revisit what they mean:

There are 228 TIA colour clocks on each scanline. 160 of those are spent drawing pixels, and 68 of them are the horizontal retrace period for the TV's scanning of the electron beam back to the start of the next line. In the diagram we see the horizontal blank (retrace) at the left side, so our very first colour clock for the TIA's first visible pixel on the screen is cycle #68. We should understand this timing fairly well by now.

What we're going to finalise this session is our understanding of the numbers down the right hand side - which represent the scanlines sent to the TV. The diagram shows a valid NTSC TV frame - and thus it consists of 262 scanlines. A PAL diagram would consist of 312 scanlines - and the inner 'picture' area would increase from 192 lines to 242 lines.

Let's go from the top. The first thing that the TV needs is a 'reset signal' to indicate to it that a new frame is starting. This is the 3-scanline section at the very top of the frame. There are special ways to trigger the TIA to send this signal, but we're not going to have to worry too much about understanding that - just about every game does it exactly the same way - all we need to remember is that the first thing to send is that reset trigger (called VSYNCH).

TVs are not all made the same. Some cut off more of the picture than others, some show wider pictures, some show taller pictures, etc. To 'standardise' the picture, the diagram shows the recommended spread of valid picture lines, surrounded by blank (or 'overscan') lines. In this case, there are 192 lines of actual picture. We don't *HAVE* to stick to this - we could steal some of the lines from the vertical blank section, and some from the overscan section, and increase our picture section appropriately.

As long as our total number of scanlines adds up to 262 for NTSC TVs (or 312 for PAL TVs), then the TV will be able to display the frame. But remember, the further we get 'out of specs' with this method, the less likely it is that ALL TVs will show the picture section in its entirety.

OK, let's march through the numbers on the right side of the diagram.

* 3 Scanlines devoted to the vertical synchronisation
* 37 scanlines of vertical blank time
* 192 (NTSC) or 242 (PAL) lines of actual picture
* 30 scanlines of overscan

Total: 262 scanlines (NTSC) or 312 scanlines (PAL), constituting a valid TV frame. You send the TV this, and it will be a rock-solid display.

One interesting aside: if you send a PAL TV an *odd* number of scanlines, it will only display in black and white. I don't know the exact reason for this, but it must be to do with where/when the colour signal is encoded in the TV image, and where the TV looks for it. So remember, always send an even number of scanlines to a PAL TV.

You *can* send frames with different numbers of scanlines. That is, 262 and 312 are not totally

immutable values. But if you do vary these numbers, it is highly likely that an increasing number of TVs - the further you deviate from these standards - will simply not be able to display your image. So, although you *can*... you shouldn't.

Fortunately, emulators available to us today are able to show us the actual number of scanlines which are being generated on each frame. This must have been quite a challenging task for early '2600 programmers - nowdays its quite easy to make sure we get it right.

Well, now we have all the knowledge we need about the composition of a TV frame. Once we know how to make the TIA generate its reset signal at the top of the frame, and how to wait the correct amount of time to allow us to correctly generate the right number of scanlines for those other sections, we will be able to design our first 'kernel' - the bit that actually 'draws' the frame.

When we have our kernel working, there's not much more to a '2600 game other than moving sprites around, changing colours, etc. See you next time.

Session 8: Our First Kernel

We're going to jump right in, now that we know what a kernel needs to do. Seen below, and in the attached file, is the source code for a working '2600 kernel. It displays the image you see here. Not bad for just a few lines of code. Over the next few sessions we'll learn how to modify this code, and assemble it - and, of course, what all those strange words mean.

For now, have a look at the structure of the code and note how closely it relates to the structure of the TV frame diagram in the earlier sessions. Don't expect to understand everything - we'll walk through every line soon. For now, all you need to know is that the "sta WSYNC" is where the 6502 is telling the TIA to halt the 6502 until the start of the next scanline. So each of those lines is where one complete scanline has been sent to the TV by the TIA. Have a close look at those lines, and see how there are 3, followed by 37 (vertical blank period), followed by 192 (picture) followed by 30 (overscan) - and how this exactly matches our TV frame diagram, above.

Yes, this is a complete kernel. It's not that difficult!

Here's the source-code...

Code:

```
processor 6502
include "vcs.h"
include "macro.h"

SEG
ORG $F000

Reset
StartOfFrame

; Start of vertical blank processing

lda #0
sta VBLANK

lda #2
sta VSYNC

; 3 scanlines of VSYNCH signal...

sta WSYNC
sta WSYNC
sta WSYNC

lda #0
sta VSYNC

; 37 scanlines of vertical blank...

sta WSYNC
sta WSYNC
```

```
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC
sta WSYNC


; 192 scanlines of picture…

ldx #0
REPEAT 192 ; scanlines

inx
stx COLUBK
sta WSYNC

REPEND


lda #%01000010
sta VBLANK ; end of screen - enter blanking
```

```
  ; 30 scanlines of overscan...

  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC
  sta WSYNC

  jmp StartOfFrame


  ORG $FFFA

  .word Reset ; NMI
  .word Reset ; RESET
  .word Reset ; IRQ

  END
```

Next session we'll have a look at how to actually assemble this code using DASM, and how to make modifications so you can play with it and test it on the emulator to see what effect your changes have.

I think all those WSYNC's look ugly so I thought I'd share this with the class

Code:

```
processor 6502
include "vcs.h"
include "macro.h"

SEG
ORG $F000

Reset
StartOfFrame

; Start of vertical blank processing

lda #0
sta VBLANK

lda #2
sta VSYNC

; 3 scanlines of VSYNCH signal...

sta WSYNC
sta WSYNC
sta WSYNC
lda #0
sta VSYNC

; 37 scanlines of vertical blank...
```

```
  REPEAT 37 ; scanlines
  sta WSYNC

  REPEND



  ; 192 scanlines of picture...

  ldx #1
  REPEAT 192 ; scanlines
  inx
  stx COLUBK
  sta WSYNC

  REPEND


  lda #%01000010
  sta VBLANK ; end of screen - enter blanking

  ; 30 scanlines of overscan...

  REPEAT 30
  sta WSYNC

  REPEND

  jmp StartOfFrame


  ORG $FFFA

  .word Reset ; NMI
  .word Reset ; RESET
  .word Reset ; IRQ

  END
```

Z26 must default to PAL because it's not using the whole screen
I changed 192 to 242 and it works fine but does the vertical blank and overscan need to be adjusted?

I welcome questions - after all, this is supposed to be an interactive tutorial/forum.

I tried to make the code sample as UNDERSTANDABLE as possible. It is certainly not the most efficient code - for it uses too many bytes of ROM to achieve its effect. But we're learning, and what's important right now is understanding how things work.

It's as good a time as any to explain a little bit about the assembler - DASM. As you have probably gathered by now, we make our changes to the source code - which is meant to be a

human-readable form of the program. We feed that source code to the assembler - and provided the assembler doesn't find any errors in the format of the code, it will convert the human-readable format into a binary format which is directly runnable on the '2600 (burn it to an EPROM, plug the EPROM into a cartridge, and plug the cartridge into a '2600) or on an emulator (just load the binary into the emulator).

Consider the following snippet of code...

Code:

```
sta WSYNC
sta WSYNC
sta WSYNC
```

That's 3 scanlines of 6502-halting. DASM has a nice feature where it can output a listing file which shows both our original source code, but also the binary numbers it replaces that code with. We'll have a close look at this feature later (and how to 'drive' DASM) - but those wishing to look through the DASM documentation should look for the "-l" switch.

When the above code fragment (from our original kernel) is assembled, the listing file contains the following...

Code:

```
25 f008 85 02 sta WSYNC
26 f00a 85 02 sta WSYNC
27 f00c 85 02 sta WSYNC
```

The leftmost number is the line-number in our original source. The next 4-digit hexadecimal number is the address in ROM of the code. Don't worry too much about that now - but do notice that each line of code is taking 2 bytes of ROM. That is, the first line starts at F008 and the next line starts at F00A (2 bytes different). That's because the "sta WSYNC" assembles to two bytes - $85 and $02. In fact, there's a 1:1 correspondence here between the mnemonic ("abbreviation") of our instruction - the human readable form - and the binary - the machine-readable form. The "sta" instruction (which stands for store-accumulator) has an opcode of $85. Whenever the 6502 fetches an instruction from ROM, and that instruction opcode is $85, it will execute the "store accumulator" instruction.

The above code fragment, then, shows three consecutive "$85 $02" pairs, corresponding exactly to our three consecutive "sta WSYNC" pairs. Can you guess the actual address of the TIA WSYNC register? If you need a clue, load up the "vcs.h" file and see what you can find in there. It should be clear to you that the assembler has simply replaced the WSYNC with an actual numerical value. To be exact, after assembling the file, it has decided that the correct value for WSYNC is 2 - and replaced all occurences of WSYNC with the number 2 in the binary image.

OK, so that was pretty straightforward - now let's do what HappyDood did, and insert that "REPEAT" thingy...

Code:

```
REPEAT 3
sta WSYNC
REPEND
```

This does do exactly the same thing, as he has surmised - but not, I suspect, quite in the way that he thinks. Let's have a look at the listing file for this one...

Code:

```
31 f008 REPEAT 3
32 f008 85 02 sta WSYNC
31 f008 REPEND
32 f00a 85 02 sta WSYNC
31 f00a REPEND
32 f00c 85 02 sta WSYNC
33 f00e REPEND
```

If you look carefully, you can see in the source code at right, we still have exactly 3 lines of code - the "sta WSYNC" code - and in the middle, we still have 3 pairs of "$85 $02" bytes in our binary. All that has changed, really, is that our source code was smaller and easier to write (especially if we're considering dozens of lines of "sta WSYNC"s.

DASM is a pretty good assembler - and it is loaded with features which make writing code easier. Happy has used one of these features to simplify the writing of the code. That feature is the "repeat" construct. Wrap any code with "REPEAT n" (where n is a number > 0), and "REPEND" and the assembler will automatically duplicate the surrounded code in the binary n times.

Note, we're not saving ROM, we're just having an easier time writing the code in the first place.

So this highlights, I hope, that it is possible to include things in your source code which are directions to the assembler - basically a guide to the assembler about how to interpret the code. REPEAT is one of those. There are several others, and we will no doubt learn about these in future sessions.

I won't introduce too much more 6502 at this stage - but what HappyDood was striving to do was simplify the code. The repeat structure was a way to do that visually, but it does not reduce ROM usage. One way (of several) to do that is to incorporate the "sta WSYNC" into a loop, which iterates 37 times. Here's a teaser...

Code:

```
; 37 scanlines of vertical blank...

ldx #0
VerticalBlank sta WSYNC
inx
cpx #37
```

```
  bne VerticalBlank
```

Remember, the 6502 has three "registers" named "X", "Y", and "A". In the code above, we initialise one register to the value 0 through "ldx #0", then we do the halt "sta WSYNC" which will halt the 6502 until the TIA finishes the current scanline. Then we increment the x-register "inx" by one, then we compare the x-register with 37 "cpx #37". This is in essence asking "have we done this 37 times yet". The final line "bne VerticalBlank" transfers control of the program back to the line "VerticalBlank" if the comparison returned (in effect) "no".

The actual listing file for that code contains the following...

Code:

```
41 f012 a2 00 ldx #0
42 f014 85 02 VerticalBlank sta WSYNC
43 f016 e8 inx
44 f017 e0 25 cpx #37
45 f019 d0 f9 bne VerticalBlank
```

If we count the number of bytes in the binary output we can see that this code takes just 9 bytes of ROM. If we had 37 "sta WSYNC" instructions, at two bytes each, that's 74 bytes of ROM. Using the REPEAT structure, as noted, will still take 74 bytes of ROM. So looping is a much more efficient way to do this sort of thing. There are even MORE efficient ways, but let's not get ahead of ourselves.

We are a bit ahead of ourselves here, so don't panic. Just remember, though, that DASM is a tool designed to aid us humans. It is full of things which make the code more readable (less "ugly") but taking lines of code out does not necessarily mean our code is more efficient - or uses less ROM

The ORG statement is one of those DASM-things which we talked about, which helps DASM to assemble the code into a binary of the correct form.

In this case, the ORG statement tells DASM what address the following code should start at. So at the beginning of the code we have ORG $F000 - which tells DASM to start putting code at $F000 (of course), and at the end we have ORG $FFFA - which telss DASM to put the next bit of code at $FFFA.

That next bit of code happens to be the "interrupt vectors". We'll cover those later, but in essence they are there to help the 6502 go to the right bit of code when it first powers-on. There are three vectors, pointing to bits of code (they contain the address of the code). When you press RESET on your '2600, the 6502 looks in location $FFFC, retrieves the two bytes starting there, and uses those bytes to form a 16-bit address where it starts running your program from.

In the case of our first sample code, that address is at $F000 - which corresponds to our "label" "Reset".

The only way to really be sure how big a ROM is, is to look at the size of the binary the assembler spits out. In our 4K ROMs, they will be 4096 bytes. In 2K ROMs they will, of course, be 2048 bytes.

2K ROMs start with a different origin...

ORG $F800

So the ROM "lives" at location $F800-$FFFF (2K in size), and the interrupt vectors are still at $FFFA onwards. Don't get too far ahead, now!

Have a look at the listing file produced with the -l option of DASM. Here's an excerpt, from a variant of our first kernel...

Code:

```
60 f02a ; 30 scanlines of overscan...
61 f02a
62 f02a a2 00 ldx #0
63 f02c 85 02 Overscan sta WSYNC
64 f02e e8 inx
65 f02f e0 1e cpx #30
66 f031 d0 f9 bne Overscan
67 f033
68 f033 4c 00 f0 jmp StartOfFrame
69 f036
70 f036
71 fffa ORG $FFFA
72 fffa
73 fffa 00 f0 .word.w Reset ; NMI
74 fffc 00 f0 .word.w Reset ; RESET
75 fffe 00 f0 .word.w Reset ; IRQ
76 10000
77 10000 END
```

Here we can see that the code itself ends at $F036. So it's using $F036 - $F000 bytes (= $36 = 54 decimal). There's also the interrupt vectors at the end - another 6 bytes. So the code sample I used to produce the above listing snippet is using just 60 of the 4096 available bytes.

Listing files are your friend.

You can use echo to report it after your ROM assembly.

For instance (using Andrew's example) the code starts at $F000. If you add the line...

Code:

```
echo *-$F000, " ROM bytes used"
```

just before org $FFFA and you assemble the code, DASM should report the number of bytes used. I use this in Climber 5 to keep track of how many bytes I have free until I reach my limit.

*-$F000 means to take the current location (*) and subtract it from the given address ($F000).

You would place this before an org or align ### statement so you get an acurate count.

Let's make it as easy as possible.

Unzip the DASM files into one directory. Also place the companion '2600 support files (from the DASM page at http://www.atari2600.org/dasm) in the same directory.

Now use a text editor to create a new file. Enter the latest source code in there (a cut/paste from your browser is fine). Now save the file as "test.asm". Exit the text editor.

Now go to a DOS command-line (or Mac equivalent).

Type "dasm test.asm -f3 -v5 -otest.bin" without the quotes.

This should cause DASM to assemble your file. If there are no errors, it will create a file called "test.bin" you can view in your emulator.

Then you repeat the process. Edit the text file "test.asm". Save it. Run DASM by typing that line again. Test the binary on the emulator. Repeat.

Much of this can be automated, by integration into an IDE like Microsoft Developer Studio. There's also a sample IDE to download/test on the DASM page.

Good luck!

PS: Here's the latest source code. Cut this, paste to "test.asm"...

Code:

```
        processor 6502
        include "vcs.h"
        include "macro.h"

;-------------------------------------------------------------
        SEG
        ORG $F000

Reset

  ; Clear RAM and all TIA registers

        ldx #0
        lda #0
Clear       sta 0,x
        inx
        bne Clear
```

```
StartOfFrame

  ; Start of vertical blank processing

        lda #0
        sta VBLANK

        lda #2
        sta VSYNC

        sta WSYNC
        sta WSYNC
        sta WSYNC       ; 3 scanlines of VSYNC signal

        lda #0
        sta VSYNC

  ; 37 scanlines of vertical blank...

        ldx #0
VerticalBlank  sta WSYNC
        inx
        cpx #37
        bne VerticalBlank


  ; 192 scanlines of picture...

        ldx #0
Picture
        SLEEP 20      ; adjust as required!

        inx
        stx COLUBK

        SLEEP 2       ; adjust as required!

        txa
        eor #$FF
        sta COLUBK

        sta WSYNC

        cpx #192
        bne Picture




        lda #%01000010
        sta VBLANK      ; end of screen - enter blanking

  ; 30 scanlines of overscan...
```

```
        ldx #0
Overscan    sta WSYNC
        inx
        cpx #30
        bne Overscan

        jmp StartOfFrame


;--------------------------------------------------------------
      ORG $FFFA

InterruptVectors

      .word Reset       ; NMI
      .word Reset       ; RESET
      .word Reset       ; IRQ

      END
```

Session 9: 6502 and DASM – Assembling the Basics

This session we're going to have a look at the assembler "DASM", what it does, how it does it, why it does it, and how to get it to do it

The job of an assembler is to convert our source code into a binary image which can be run by the 6502. This conversion process ultimately replaces the mnemonics (the words representing the 6502 instructions we use when writing in assembler) and the symbols (the various names we use for things, such as labels to which we can branch, and various other things like the names of TIA registers, etc) with numerical values.

So ultimately, all the assembler needs to do is figure out a numerical value for all the things which become part of the binary - and place that value in the appropriate place in the binary.

We've already had a brief introduction to a 6502 instruction - the one called "nop". This is the no-operation instruction which simply takes 2 cycles to execute. Whenever we enter "nop" into our source code, the assembler recognises this as a 6502 instruction and inserts into the binary the value $EA. This shows that there can be a simple 1:1 relationship between source-code and the binary.

"nop" is a single-byte instruction - all it requires is the opcode, and the 6502 will happily execute it. Some instructions require additonal "parameters" - the "operands". The 6502 microprocessor can use an additional 1 or 2 bytes of operand data for some instructions, so the total number of bytes for a 6502 "instruction" can be 1, 2 or 3.

DASM is the assembler used by most (if not all) modern-day '2600 programmers. It is a multi-platform assembler written in 1988 by Matt Dillon (you should all find his email address and send him a "thank-you" sometime). It's a great tool.

DASM isn't just capable of assembling 6502 (and variant) code - it also has inbuilt capability to assemble code for several other microprocessors. Consequently, one of the very first things that it is necessary to do in our source code is tell DASM what processor the source code is written for.

Code:

```
processor 6502
```

This should be just about the first line in any '2600 program you write. If you don't include it, DASM will probably get confused and spit out errors. That's simply because it is trying to assemble your code as if it were written for another processor.

We've just seen how mnemonics (the standard names for instructions) are converted into numerical values by the assembler. Another job the assembler does is convert labels and symbols into values. We've already encountered both of these in our previous sessions, but you may not be familiar with their names.

Whenever DASM is doing its job assembling, it keeps a list of all the "words" it encounters in a file in an internal structure called a symbol table. Think of a symbol as a name for something. Remember the "sta WSYNC" instruction we used to halt the 6502 and wait for the scanline to be rendered? The "sta" is the instruction, and "WSYNC" is a symbol. When it first encounters this

symbol, DASM doesn't know much about it, other than what it's called (i.e.: "WSYNC"). What DASM needs to do is work out what the *value* of that symbol is, so that it can insert that value into the binary file.

When it's assembling, DASM puts all the symbols it finds into its symbol table - and associated with each of these is a value. If it doesn't "know" the value, that's OK - DASM will keep assembling the rest of the file quite happily. At some point, something in the code might tell DASM what the value for a symbol actually IS - in which case DASM will put that value in its symbol table alongside the symbol. So whenever that symbol is used anywhere, DASM now knows its correct value to put into the binary file.

In fact, it is absolutely necessary for all symbols which go into the binary file to be given values at some point. DASM can't guess values - it's up to you, the programmer, to make sure this happens. A symbol doesn't have to be given a value at any PARTICULAR point in the code, but it does have to be given a value somewhere in the code. DASM will make multiple "passes" - basically going through the code from beginning to end again and again until it manages to resolve all the symbols to correct values.

We've already seen in some sample code how "sta WSYNC" appears in our binary file as the bytes $85 $02. The first byte $85 is the "sta" instruction (one variant of many - but let's keep it simple for now) and it is followed by a single byte giving the address of the location into which the byte in the "A" register is to be stored. We can see this address is location 2 in memory. Somehow, DASM has figured out from the code that the symbol WSYNC has a value of 2, and when it creates the binary file it replaces all occurences of the symbol with the numeric value 2.

How did it get the value 2? Remember, WSYNC is one of the TIA registers. It appears to the 6502 as a memory location, as the TIA registers are "mapped" into locations 0 - $7F. The file "vcs.h" defines (in a roundabout way) the values and names (symbols) for all of the TIA registers. By including the file "vcs.h" as a part of the assembly for any source file, we automatically tell DASM the correct numeric value for all of the TIA register "names".

That's why, at the top of most files, just after the processor statement, we see...

Code:

```
include "vcs.h"
```

You don't really need to know much about vcs.h at this stage - but be aware that a "standardised" version of this file is distributed with the DASM assembler as the '2600 support files package. I would advise you to always use the latest and greatest version of this file. Standards help us all.

So now we know basically what DASM does with symbols - it keeps an internal list of symbols - and their values, if known. DASM will keep going through the code and "resolving" the symbols into numeric values, until it is complete (or it couldn't find ANYTHING to resolve, in which case it gives an error). Once all symbols have been resolved, your code has been completely processed by the assembler, and it creates the binary image/file for you - and assembly is complete.

To summarise: DASM converts source-code consisting of instructions (mnemonics) and symbols into a binary form which can be run by the 6502. The assembler converts mnemonics into opcodes (numbers), and symbols into numbers which it calculates the value of during the

assembly process.

DASM is a command-line program - that is, it runs under DOS (or whatever platform you happen to choose, provided you have a runnable version for that platform). DASM is provided with full source-code (it's written in C) so as long as you have a C-compiler handy, you can port it to just about any platform under the sun.

It does come with a manual - and it's always a good idea to familiarise yourself with its capabilities. In the interests of getting you up and running quickly, so you can actually assemble the sample kernel posted a session or two ago, here's what you need to type on the command-line...

Code:

```
dasm kernel.asm -lkernel.txt -f3 -v5 -okernel.bin
```

This is assuming that the file to assemble is named "kernel.asm" (.asm is a standard prefix for assembler files, but some prefer to use .s - you can use whatever you want, really, but I always use .asm). Anything prefixed with a minus-sign ("-") is a "switch" - which tells DASM something about what it is required to do. The -l switch we discussed very briefly, and that tells DASM to create a listing file - in this case, it will write a listing to the file "kernel.txt". The -o switch tells DASM what file to use for the output binary - in this case, the binary will be written to "kernel.bin". That file can be loaded into an emulator, or burned on an EPROM - it is the ROM file, in other words.

The other switches "-f3" and "-v5" control some internals of DASM - and for now just assume you need these whenever you assemble with DASM. Remember, if you're curious you can always read the manual!

If all goes well, DASM will output something like this...

Code:

```
DASM V2.20.05, Macro Assembler (C)1988-2003
START OF PASS: 1
--------------------------------------------------------------------------
SEGMENT NAME INIT PC INIT RPC FINAL PC FINAL RPC
f000 f000
RIOT [u] 0280 0280
TIA_REGISTERS_READ [u] 0000 0000
TIA_REGISTERS_WRITE [u] 0000 0000
INITIAL CODE SEGMENT 0000 ???? 0000 ????
--------------------------------------------------------------------------
1 references to unknown symbols.
0 events requiring another assembler pass.
--- Symbol List (sorted by symbol)
AUDC0 0015
AUDC1 0016
AUDF0 0017
AUDF1 0018
AUDV0 0019
```

AUDV1 001a
COLUBK 0009 (R)
COLUP0 0006
COLUP1 0007
COLUPF 0008
CTRLPF 000a
CXBLPF 0006
CXCLR 002c
CXM0FB 0004
CXM0P 0000
CXM1FB 0005
CXM1P 0001
CXP0FB 0002
CXP1FB 0003
CXPPMM 0007
ENABL 001f
ENAM0 001d
ENAM1 001e
GRP0 001b
GRP1 001c
HMBL 0024
HMCLR 002b
HMM0 0022
HMM1 0023
HMOVE 002a
HMP0 0020
HMP1 0021
INPT0 0008
INPT1 0009
INPT2 000a
INPT3 000b
INPT4 000c
INPT5 000d
INTIM 0284
NUSIZ0 0004
NUSIZ1 0005
Overscan f02c (R)
PF0 000d
PF1 000e
PF2 000f
Picture f01d (R)
REFP0 000b
REFP1 000c
RESBL 0014
Reset f000 (R)
RESM0 0012
RESM1 0013
RESMP0 0028
RESMP1 0029
RESP0 0010
RESP1 0011
RSYNC 0003
StartOfFrame f000 (R)

```
SWACNT 0281
SWBCNT 0283
SWCHA 0280
SWCHB 0282
T1024T 0297
TIA_BASE_ADDRESS 0000 (R )
TIM1T 0294
TIM64T 0296
TIM8T 0295
TIMINT 0285
VBLANK 0001 (R )
VDELBL 0027
VDELP0 0025
VDELP1 0026
VerticalBlank f014 (R )
VSYNC 0000 (R )
WSYNC 0002 (R )
--- End of Symbol List.
Complete.
```

Here we can actually SEE the symbol table, and the numeric values that DASM has assigned to the symbols. If you look at the listing file, wherever any of these symbols is used, you will see the corresponding number in the symbol table has been inserted into the binary.

There are lots of symbols there, as the vcs.h file defines just about everything you'll ever need to do with the TIA. The symbols which are actually USED in your code are marked with a (R) - indicating "referenced".

Now you should be able to go and assemble the sample kernel I provided earlier. Don't be afraid to have a play with things, and see what happens! Experimenting is a big part of learning.

Soon we'll start playing with some TIA registers and seeing what happens to our screen when we do that! For now, though, make sure you are able to assemble and run the first kernel. If you have any problems, ask for assistance and I'm sure somebody will leap to your aid.

By default I include "vcs.h" and "macro.h" files in all source code. These are standardised files for '2600 development, and distributed as official DASM '2600 support files.

MACROs are a sort of text-processing language supported by DASM. In the same way that the REPEAT keyword allowed us to repeat blocks of code automatically, MACROs allow us to package common functionality into a single keyword and have the assembler insert (and tailor) code automatically.

There's nothing in the macro.h file we use, yet... but it is good practise to include it - as it has some useful content already, and will have more added from time to time.

As a teaser, consider the SLEEP macro... remember how we wanted to delay 76 cycles for each scanline, and we used the "sta WSYNC" capability of the TIA to halt the 6502 till the start of the next scanline? Or how we used NOP to waste exactly 2 cycles. Use the sleep macro to delay for any number of cycles you want... e.g.:

```
SLEEP 25 ; waste 25 cycles.
```

The SLEEP macro is defined in macro.h, if you want to see how it does it.

Session 10: Orgasm

We've had a brief introduction to DASM, and in particular mnemonics (6502 instructions, written in human-readable format) and symbols (other words in our program which are converted by DASM into a numeric form in the binary).

Now we're going to have a brief look at how DASM uses the symbols (and in particular the value for symbols it calculates and stores in its internal symbol table) to build up the binary ROM image.

Each symbol the assembler finds in our source code must be defined (i.e.: given an actual value) in at least one place in the code. A value is given to a symbol when it appears in our code starting in the very first column of a line. Symbols typically cannot be redefined (given another value).

In an earlier session we examined how the code "sta WSYNC" appeared in our binary file as $85 $02 (remember, we examined the listing file to see what bytes appeared in our binary. At that point, I indicated that the assembler had determined the value of the symbol "WSYNC" was 2 (corresponding to the TIA register's memory address) - through its definition in the standard vcs.h file.

But how does the assembler actually determine the value of a symbol?

The answer is that the symbol must be defined somewhere in the source code (as opposed to just being referenced). Definition of a symbol can come in several forms. The most straightforward is to just assign a value...

Code:

```
WSYNC = 2
```

or...

Code:

```
WSYNC EQU 2
```

The above examples are equivalent - DASM supports syntax (style) which has become fairly standard over the years. Some people (me!) like to use the = symbol, and some like to use EQU. Note that the symbol in question must start in the very first column, when it is being defined. In both cases, the value 2 is being assigned to the symbol WSYNC. Wherever DASM encounters the symbol WSYNC in the code, it knows to use the value 2.

That's fairly straightforward stuff. But symbols can be defined in terms of other symbols! Also, DASM has a quite capable ability to understand expressions, so the following is quite valid...

Code:

```
AFTER_WSYNC = WSYNC + 1
```

In this case, the symbol "AFTER_WSYNC" would have the value 3. Even if the WSYNC label was defined after the above code, the assembler would successfully be able to resolve the AFTER_WSYNC value, as it does multiple passes through the code until symbols are all resolved.

Symbols can also be given values automatically by the assembler. Consider our sample kernel where we see the following code near the start (here we're looking at the listing file, so we can see the address information DASM outputs)...

Code:

```
10 0000 ???? SEG
11 f000 ORG $F000
12 f000
13 f000 Reset
14 f000
15 f000
16 f000
17 f000
18 f000
19 f000
20 f000 StartOfFrame
21 f000
22 f000 ; Start of vertical blank processing
23 f000
24 f000 a9 00 lda #0
25 f002 85 01 sta VBLANK
```

"Reset" and "StartOfFrame" are two symbols which are definitions at this point because they both start at the first column of the lines they are on. The assembler assigns the current ROM address to these symbols, as they occur. That is, if we look at these "labels" (=symbols) in the symbol table, we see...

Code:

```
StartOfFrame f000 (R )
Reset f000 (R )
```

They both have a value of $F000. This form of symbol (which starts at the beginning of a line, but is not explicitly assigned a value) is called a label, and refers to a location in the code (or more particularly an address). How and why did DASM assign the value $F000 to these two labels, in this case?

As the assembler converts your source code to a binary format, it keeps an internal counter telling it where in the address space the next byte is to be placed. This address increments by

the appropriate amount for each bit of data it encounters. For example, if we had a "nop" (a 1-byte instruction), then the address counter that DASM maintains would increment by 1 (the length of the nop instruction). Whenever a label is encountered, the label is given the value of the current internal address counter at the point in the binary image at which the label occurs. The label itself does not go into the binary - but the value of the label refers to the address in the binary corresponding to the position of the label in the source code.

In the above code snippet, we can see the address in column 2 of the output, and it starts at 0 (with ???? after it, indicating it doesn't actually KNOW the internal counter/address at this point), and (here's the bit I really want you to understand) it is set to $F000 when we get the "org $F000" line. "Org" stands for origin, and this is the way we (the programmer) indicate to the assembler the starting address of next section of code in the binary ROM. Just to complicate things slightly, it is not the actual offset from the start of the ROM (for a ROM might, for example, be only 4K but contain code assembled to live at $F000-$FFFF - as in a 4K cartridge). So it's not an offset, it's a conceptual address.

These labels are very useful to programmers to give a name to a point in code, so that that point may be referred to by the label, instead of us having to know the address. If we look at the end of our sample kernel, we see...

Code:

```
70 f3ea 4c 00 f0 jmp StartOfFrame
```

The "jmp" is the mnemonic for the jump instruction, which transfers flow of control to the address given in the two byte operand. In other words, it's a GOTO statement. Look carefully at the binary numbers inserted into the ROM (again, the columns are left to right, line number, address, byte(s), source code). We see $4C, 0, $f0. The opcode for JMP is $4C - whenever the 6502 fetches this instruction, it forms a 16-bit address from the next two bytes (0,$F0) and code continues from that address. Note that the "StartOfFrame" symbol/label has a value $F000 in our symbol table.

It's time to understand how 16-bit numbers are formed from two 8-bit numbers, and how 0, $F0 translates to $F000. The 6502, as noted, can address 2^16 bytes of memory. This requires 16 bits. The 6502 itself is only capable of manipulating 8-bit numbers. So 16-bit numbers are stored as pairs of bytes. Consider any 16-bit address in hexadecimal - $F000 is convenient enough. The binary value for that is %1111000000000000. Divide it into two 8-bit sections (ie: equivalent to 2 bytes) and you get %11110000 and %00000000 - equivalent to $F0 and 0. Note, any two hex digits make up a byte, as hex digits require 4 bits each (0-15, ie: %0000-%1111). So we could just split any hex address in half to give us two 8-bit bytes. As noted, 6502 manipulates 16-bit addresses through the use of two bytes. These bytes are generally always stored in ROM in little-endian format (that is, the lowest significant byte first, followed by the high byte). So $F000 hex is stored as 0, $F0 (the low byte of $F000 followed by the high byte).

Now the binary of our jmp instruction should make sense. Opcode ($4C), 16-bit address in low/high format ($F000). When this instruction executes, the program jumps to and continues executing from address $F000 in ROM. And we can see how DASM has used its symbol table - and in particular the value it calculated from the internal address counter when the StartOfFrame label was defined - to "fill in" the correct low/hi value into the binary file itself where the label was actually referred to.

This is typical of symbol usage. DASM uses its internal symbol table to give it a value for any symbol it needs. Those values are used to create the correct numbers for the ROM/binary image.

Let's go back to our magical discovery that the "org" instruction is just a command to the assembler (it does not appear in the binary) to let the assembler know the value of the internal address counter at that point in the code. It is quite legal to have more than one ORG command in our source. In fact, our sample kernel uses this when it defines the interrupt vectors...

Code:

```
70 f3ea 4c 00 f0 jmp StartOfFrame
71 f3ed
72 f3ed
73 fffa ORG $FFFA
74 fffa
75 fffa 00 f0 .word.w Reset ; NMI
76 fffc 00 f0 .word.w Reset ; RESET
77 fffe 00 f0 .word.w Reset ; IRQ
```

Here we can see that after the jmp instruction, the internal address counter is at $F3ED, and we have another ORG which sets the address to $FFFA (the start of the standard 6502 interrupt vector data). Astute readers will notice the use of the label "Reset" in three lines, with the binary value $F000 (if the numbers are to be interpreted as a low/high byte pair) appearing in the ROM image at address $FFFA, $FFFC, $FFFE. We briefly discussed how the 6502 looks at the address $FFFC to give it the address at which it should start running code. Here we see that this address points to the label "Reset". Magic.

It's quite legal to use one symbol as the value for an ORG command. Here's a short snippet of code which should clarify this...

Code:

```
START = $F800 ; start of code - change this if you want

ORG START
HelloWorld
```

In the above example, the label HelloWorld would have a value of $F800. If the value of START were to change, so would the value of HelloWorld.

We've seen how the ORG command is used to tell DASM where to place bits of code (in terms of the address of code in our ROM). This command can also be used to define our variables in RAM. We haven't had a play with RAM/variables yet, and it will be a few sessions before we tackle that - but if you want a sneek peek, have a look at vcs.h and see how it defines its variables from an origin defined as "ORG TIA_BASE_ADDRESS". That code is way more complex than our current

level of understanding, but it gives some idea of the versatility of the assembler.

We're almost done with the basic commands inserted into our source code to assist DASM's building of the binary image. Now you should understand how symbols are assigned values (either by their explicit assignation of a value, or by implicit address/location value) - and how those values - through the assembler's internal symbol table - are used to put the correct number into the ROM binary image. We also understand that DASM converts mnemonics (6502 commands in human-readable form) directly into opcodes. There's not much more to actual assembly - so we shall soon move on to actual 6502 code, and playing with the TIA itself.

Session 11: Colourful Colours

Even our language treats "color" differently - here in Oz we write "colour" and in the USA they write "color". Likewise, '2600 units in different countries don't quite speak the same language when it comes to colour.

We have already seen why there are 3 variants of '2600 units - these variations (PAL, NTSC, SECAM) exist because of the differences in TV standards in various countries. Specifically, the colour information is encoded in different ways into the analogue TV signal for each system, and the '2600 hardware is responsible for inserting that colour information in the data sent to the TV.

Not only do these different '2600 systems write the colour information in different ways, they also write totally different colours! What is one colour on a NTSC system is probably NOT the same colour on PAL, and almost certainly not the same colour on SECAM! Here's some wonderful colour charts from B. Watson to show the colours used by each of the systems...

http://www.urchlay.com/stelladoc/v2/tia_colorchart.html

Colours are represented on the '2600 by numbers. How else could it be? The colour to number correspondence is essentially an arbitrary association - so, for example on a NTSC machine the value $1A is yellowish, on PAL the same colour is grey, and on SECAM it is aqua (!). If the same colour values were used on a game converted between a NTSC and PAL system, then everything would look very weird indeed! To read the colour charts on the above URL, form a 2-digit hex number from the hue and the lum values (ie: hue 2, lum 5 -> $25 value -> brown(ish) on NTSC and as it happens a very similar brown(ish) on PAL.

We've already played with colours in our first kernel! In the picture section (the 192 scanlines) we had the following code...

Code:

```
; 192 scanlines of picture...

ldx #0
REPEAT 192 ; scanlines

inx
stx COLUBK
sta WSYNC

REPEND
```

We should know by now what that "sta WSYNC" does - and now it's time to understand the rest of it. Remember the picture that the kernal shows? A very pretty rainbow effect, with colour stripes across the screen. It's the TIA producing those colours, but it's our kernal telling the TIA what colour to show on each line. And it's done with the "stx COLUBK" line.

Remember how the TIA maps to memory in locations 0 - $7F, and that WSYNC is a label

representing the memory location of the TIA register (which happens, of course, to be called WSYNC). In similar fashion, COLUBK is a label which corresponds to the TIA register of the same name. This particular register allows us to set the colour of the background that the TIA sends to the TV!

A quick peek at the symbol table shows...

Code:

```
COLUBK 0009 (R )
```

In fact, the very best place to look is in the Stella Programmer's guide - for here you will be able to see the exact location and usage of this TIA register. This is a pretty simple one, though - all we do is write a number representing the colour we want (selected from the colour charts linked to, above) and the TIA will display this colour as the background.

Don't forget that it also depends on what system we're running on! If we're doing a PAL kernel, then we will see a different colour than if we're doing a NTSC or SECAM kernel. The bizarre consequence of this is that if we change the number of scanlines our kernel generates, the COLOURS of everything also change. That's because (if we are running on an emulator or plug a ROM into a console) we are essentially switching between PAL/NTSC/SECAM systems, and as noted these systems send different colour information to the TV! It's weird, but the bottom line is that when you choose colours, you choose them for the particular TV standard you are writing your ROM to run on. If you change to a different TV system, then you will also need to rework all the colours of all the objects in your game.

Let's go back to our kernel and have a bit of a look at what it's doing to achieve that rainbow effect. There's remarkably little code in there for such a pretty effect.

As we've learned, the 6502 has just three "registers". These are named A, X and Y - and allow us to shift bytes to and from memory - and perform some simple modifications to these bytes. In particular, the X and Y registers are known as "index registers", and these have very little capability (they can be loaded, saved, incremented and decremented). The accumulator (A) is our workhorse register, and it is this register used to do just about all the grunt-work like addition, subtraction, and bit manipulation.

Our simple kernel, though, uses the X register to step a colour value from 0 (at the start), writing the colour value to the TIA background colour register (COLUBK), incrementing X by one each scanline. First (outside the repeat) we have "ldx #0". This instruction moves the numeric value 0 into the X register. ld is an abbreviation for "load", and we have lda, ldx, ldy. st is the similar abbreviation for store, and we have stx sty sta. Inside our repeat structure, we have "stx COLUBK". As noted, this will copy the current contents of the x register into the memory location 9 (which is, of course, the TIA register COLUBK). The TIA will then *immediately* use the value we wrote as the background colour sent to the TV. Next we have an instruction "inx". This increments the current value of the X register by one. Likewise, we have an "iny" instruction, which increments the y register. But, alas, we don't have an "ina" instruction to increment the accumulator (!). We are also able to decrement (by 1) the x and y registers with "dex" and "dey".

The operation of our kernel should be pretty obvious, now. The X register is initialised with 0, and every scanline it is written to the background colour register, and incremented. So the background colour shows, scanline by scanline, the colour range that the '2600 is capable of. In

actual fact, you could throw another "inx" in there and see what happens. Or even change the "inx" to "dex" - what do you think will happen? As an aside, it was actually possible to blow up one early home computer by playing around with registers like this (I kid you not!) - but you can't possibly damage your '2600 (or emulator!) doing this. Have fun, experiment.

Since we're only doing 192 lines, the X register will increment from 0 to 192 by the time we get to the end of our block of code. But what if we'd put two "inx" lines in? We'd have incremented the X register by 192 x 2 = 384 times. What would its value be? 384? No - because the X register is only an 8-bit register, and you would need 9 bits to hold 384 (binary %110000000). When any register overflows - or is incremented or decremented past its maximum capability, it simply "wraps around". For example, if our register had %11111111 in it (255, the maximum 8-bit number) and it was incremented, then it would simply become %00000000 (which is the low 8-bits of %100000000). Likewise, decrementing from 0 would leave %11111111 in the register. This may seem a bit confusing right now, but when we get used to binary arithmetic, it will seem quite natural. Hang in there, I'll avoid throwing the need to know this sort of stuff at you for a while.

Now you've had a little introduction to the COLUBK register, I'd just like to touch briefly on the difference apparent between the WSYNC register and the COLUBK register. The former (WSYNC) was a strobe - you could simply "touch" it (by writing any value) and it would instantly halt the 6502. Didn't matter what value you wrote, the effect was the same. The latter register (COLUBK) was used to send an actual VALUE to the TIA (in this case, the value for the colour for the background) - and the value written was very much important. In fact, this value is stored internally by the TIA and it keeps using the value it has internally as the background colour until it changes.

If you think about the consequences of this, then, the TIA has at least one internal memory location which is in an uknown (at least by us) state when the machine first powers on. We'd probably see black - which happens to be value 0 on all machines), but you never know. I believe it is wise to initialise the TIA registers to known-states when your kernel first starts - so there are no surprises on weird machines or emulators. We have done nothing, so far, to initialise the TIA - or the 6502, for that matter - and I think we'll probably have a brief look at system startup code in a session real-soon-now.

Until then, have a play with the picture-drawing section, and see what happens when you write different values to the COLUBK register. You might even like to change it several times in succession and see what happens. Here's somtehing to try (with a bit of headscratching, you should be able to figure all this out by now)...

Code:

```
; 192 scanlines of picture...

ldx #0
ldy #0
REPEAT 192 ; scanlines

inx
stx COLUBK
```

```
  nop
  nop
  nop

  dey
  sty COLUBK

  sta WSYNC

  REPEND
```

Try inserting more "nop" lines (what does nop do, again?) - can you see how the timing of the 6502 and where you do changes to the TIA is reflected directly onscreen because of the synchronisation between the 6052 and the TIA which is drawing the lines on-the-fly?

Have a good play with this, because once you've cottoned-on to what's happening here, you will have no problems programming anything on the '2600.

Alright then NOP = waste 1 cycle
1 6502 cycle = 3 TIA clocks

Is 160 cycles resolution the 6502 time or TIA time?

Almost. NOP = waste 2 cycles.

160 is the number of clocks of visible pixels. That's TIA clocks, one clock per pixel. But on each scanline there's 228 clocks. Divide by 3 to get the number of 6502 clocks per scanline = 76.

Andrew Davie wrote:

Until then, have a play with the picture-drawing section, and see what happens when you write different values to the COLUBK register. You might even like to change it several times in succession and see what happens. Here's somtehing to try (with a bit of headscratching, you should be able to figure all this out by now)...

Code:

```
      ; 192 scanlines of picture...

      ldx #0
      ldy #0
      REPEAT 192 ; scanlines
```

```
        inx
        stx COLUBK

        nop
        nop
        nop

        dey
        sty COLUBK

        sta WSYNC

    REPEND
```

Try inserting more "nop" lines (what does nop do, again?) - can you see how the timing of the 6502 and where you do changes to the TIA is reflected directly onscreen because of the synchronisation between the 6052 and the TIA which is drawing the lines on-the-fly?

Have a good play with this, because once you've cottoned-on to what's happening here, you will have no problems programming anything on the '2600.

See you next time!

The above may not have been so obvious, because ALL of the code inside our REPEAT... REPEND section is happening in the first 68 colour-clocks of the line, when the TV is doing its vertical retrace. If you replace the code with the following (with more NOPs inserted) the effect I was trying to show will become WAY more apparent! The sample image shows what we're looking at.

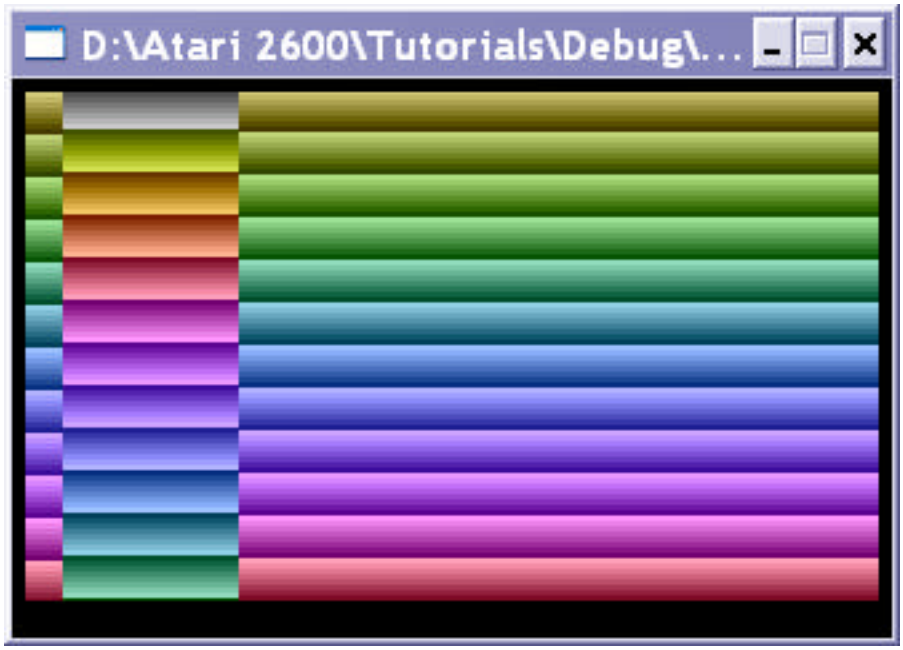
Code:

```
        ; 192 scanlines of picture...

        ldx #0
        ldy #0
        REPEAT 192 ; scanlines

            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
```

```
    inx
    stx COLUBK

    nop
    nop
    nop

    dey
    sty COLUBK

    sta WSYNC

REPEND
```

One caution: as the above code is wrapped inside a repeat structure which creates 192 copies of the enclosed code, we're actually running short of ROM space! With the above code installed, there's only 10 bytes free in our entire ROM! Clearly, using REPEAT in this sort of situation is wasteful, and the code should be written as a loop. We covered looping for scanline draw early on - but because both X and Y registers are in use at the moment, it's a bit more tricky.

So for now, we'll just have to accept that we can't add any more code - but at least you can see what effect adding/removing cycles can have on the existing code.



How many cycles does the "bit VSYNC" used in macro.h waste ?

It uses 3. There are various ways of 'efficiently' wasting time (ie: the most cycles used for the least number of bytes of ROM spent doing it). We'll cover that in a later tutorial.

If you *have* to waste time, then do it in as few bytes as possible.

The following contrived code snippets all waste the same amount of time - but the cost of the 2nd (in ROM bytes used) is double.

Code:

```
jsr _rts ; 12 cycles, 3 bytes
```

Code:

```
nop
nop
nop
nop
nop
nop ; 12 cycles, 6 bytes
```

We will see in later tutorials where, even if you DO have lots of ROM space, it is often necessary to keep code size to a minimum - because of constraints on how far the 6502 can "see" to get to other bits of code. We'll cover all of this in future sessions.

Session 12: Initialization

One of the joys of writing '2600 programs involves the quest for efficiency - both in processing time used, and in ROM space required for the code. Every now and then, modern-day '2600 programmers will become obsessed with some fairly trivial task and try to see how efficient they can make it.

If you were about to go up on the Space Shuttle, you wouldn't expect them to just put in the key, turn it on, and take off. You'd like the very first thing they do is to make sure that all those switches are set to their correct positions. When our Atari 2600 (which, I might point out in a tenuous link to the previous sentence, is of the same vintage as the Space Shuttle) powers-up, we should assume that the 6502, RAM and TIA (and other systems) are in a fairly uknown state. It is considered good practise to initialise these systems. Unless you really, *really* know what you're doing, it can save you problems later on.

At the end of this session I'll present a highly optimised (and best of all, totally obscure piece of code which manages to initialise the 6502, all of RAM *and* the TIA using just 9 bytes of code-size. That's quite amazing, really. But first, we're going to do it the 'long' way, and learn a little bit more about the 6502 while we're doing it.

We've already been introduced to the three registers of the 6502 - A, X, and Y. X and Y are known as index registers (we'll see why, very soon) and A is our accumulator - the register used to do most of the calculations (addition, subtraction, etc).

Let's have a look at the process of clearing (writing 0 to) all of our RAM. Our earlier discussions of the memory architecture of the 6502 showed that the '2600 has just 128 bytes ($80 bytes) of RAM, starting at address $80. So, our RAM occupies memory from $80 - $FF inclusive. Since we know how to write to memory (remember the "stx COLUBK" we used to write a colour to the TIA background colour register), it should be apparent that we could do this...

Code:

```
lda #0 ; load the value 0 into the accumulator
sta $80 ; store accumulator to location $80
sta $81 ; store accumulator to location $81
sta $82 ; store accumulator to location $82
sta $83 ; store accumulator to location $83
sta $84 ; store accumulator to location $84
sta $85 ; store accumulator to location $85

; 119 more lines to store 0 into location $86 - $FC...

sta $FD ; store accumulator to location $FD
sta $FE ; store accumulator to location $FE
sta $FF ; store accumulator to location $FF
```

You're right, that's ugly! The code above uses 258 bytes of ROM (2 bytes for each store, and 2

for the initial accumulator load). We can't possibly afford that - and especially since I've already told you that it's possible to initialise the 6502 registers, clear RAM, *AND* initialise the TIA in just 9 bytes total!

The index registers have their name for a reason. They are useful in exactly the situation above, where we have a series of values we want to read or write to or from memory. Have a look at this next bit of code, and we'll walk through what it does...

Code:

```
ldx #0
lda #0
ClearRAM sta $80,x
inx
cpx #$80
bne ClearRAM
```

Firstly, this code is nowhere-near efficient, but it does do the same job as our first attempt and uses only 11 bytes. It achieves this saving by performing the clear in a loop, writing 0 (the accumulator) to one RAM location every iteration. The key is the "sta $80,x" line. In this "addressing mode", the 6502 adds the destination address ($80 in this example - remember, this is the start of RAM) to the current value of the X register - giving it a final address - and uses that final address as the source/destination for the operation.

We have intialised X to 0, and increment it every time through the loop. The line "cpx #$80" is a comparison, which causes the 6502 to check the value of X against the number $80 (remember, we have $80 bytes of RAM, so this is basically saying "has the loop done 128 ($80) iterations yet?". The next line "bne ClearRAM" will transfer program flow back to the label "ClearRAM" every time that comparison returns "no". The end result being that the loop will iterate exactly 128 times, and that the indexing will end up writing to 128 consecutive memory locations starting at $80.

Code:

```
ldx #$80
lda #0
ClearRAM
sta 0,x
inx
bne ClearRAM
```

Well, that's not a LOT different, but we're now using only 9 bytes to clear RAM - somehow we've managed to get rid of that comparison! And how come we're writing to 0,x not $80,x? All will be

revealed...

When the 6502 performs operations on registers, it keeps track of certain properties of the numbers in those registers. In particular, it has internal flags which indicate if the number it last used was zero or non-zero, positive or negative, and also various other properties related to the last calculation it did. We'll get to all of that later. All of these flags are stored in an 8-bit register called the "flags register". We don't have easy direct access to this register, but we do have instructions which base their operation on the flags themselves.

We've already used one of these operations - the "bne ClearRAM" we used in our earlier version of the code. This instruction, as noted "will transfer program flow back to the label "ClearRAM" every time that comparison returns "no". The comparison returns "no" by setting the zero/non-zero flag in the processor's flags register!

In atuality, this zero/non-zero flag is also set or cleared upon a load to a register, an increment or decrement of register or memory, and whenever a calculation is done on the accumulator. Whenever a value in these circumstances is zero, then the zero flag is set. Whenever the result is non-zero, the zero flag is cleared. So, we don't even need to compare for anything being 0 - as long as we have just done one of the operations mentioned (load, increment, etc) - then we know that the zero flag (and possibly others) will tell us something about the number. The 6502 documentation gives extensive information for all instructions about what flags are set/cleared, under what circumstance.

We briefly discussed how index registers, only holding 8-bit values "wrap-around" from $FF (%11111111) to 0 when incremented, and from 0 to $FF when decremented. Our code above is using this "trick" by incrementing the X-register and using the knowledge that the zero-flag will always be non-zero after this operation, unless X is 0. And X will only be 0 if it was previously $FF. Instead of having X be a "counter" to give 128 iterations, this time we're using it as the actual address and looping it from $80 (the start of RAM) to $FF (the end of RAM) + 1. SO our store (which, remember, takes the address in the instruction, adds the value of the X register and uses that as the final address) is now "sta 0,x". Since X holds the correct address to write to, we are adding 0 to that

I would *highly* recommend that you don't worry too much about this sort of optimisation while you're learning. The version with the comparison is perfectly adequate, safe, and easy to understand. But sometimes you find that you do need the extra cycles or bytes (the optimised version, above, is 160 cycles faster - and that's 160x3 colour clocs = 480 colour clocks = more than two whole scanlines !! quicker). So you can see how crucial timing can be - by taking out a single instruction (the "cpx #$80") in a loop, and rearranging how our loop counted, we saved more than two scanlines - (very) roughly 1% of the total processing time available in one frame of a TV picture!

Initialising the TIA is a similar process to initialising the RAM - we just want to write 0 to all memory locations from 0 to $7F (where the TIA lives!). This is safe - trust me - and we don't really need to know what we're writing to at this stage, just that after doing this the TIA will be nice and happy. We could do this in a second loop, similar to the first, but how about this...

Code:

```
ldx #0
lda #0
Clear
sta $80,x ; clear a byte of RAM
```

```
sta 0,x ; clear a byte of TIA register
inx
cpx #$80
bne Clear
```

That's a perfectly adequate solution. Easy to read and maintain, and reasonably quick. We could, however, take advantage of the fact that RAM and the TIA are consecutive in memory (TIA from 0 - $7F, immediately followed by RAM $80 - $FF) and do the clear in one go...

Code:

```
ldx #0
lda #0
Clear
sta 0,x
inx
bne Clear
```

The above example uses 9 bytes, again, but now clears RAM and TIA in one 'go' by iterating the index register (which is the effective address when used in "sta 0,x") from 0 to 0 (ie: increments 256 times and then wraps back to 0 and the loop halts). This is starting to get into "elegant" territory, something the experienced guys strive for!

Furthermore, after this code has completed, X = 0 and A = 0 - a nice known state for two of the 3 6502 registers.

That's all I'm going to explain for the initialisation at this stage - we should insert this code just after the "Reset" label and before the "StartOfFrame" label. This would cause the code to be executed only on a system reset, not every frame (as, every frame, the code branches back to the "StartOfFrame" for the beginning of the next frame).

Before we end today's session, though, I thought I'd share the "magical" 9-byte system clear with you. There's simply no way that I would expect you to understand this bit of code at the moment - it pulls every trick in the book - but this should give you some taste of just how obscure a bit of code CAN be, and how beautifully elegant clever coding can do amazing things.

Code:

```
; CLEARS ALL VARIABLES, STACK
; INIT STACK POINTER
; ALSO CLEARS TIA REGISTERS
; DOES THIS BY "WRAPPING" THE STACK - UNUSUAL

LDX #0
TXS
PHA ; BEST WAY TO GET SP=$FF, X=0

TXA
CLEAR PHA
```

```
DEX
BNE CLEAR

; 9 BYTES TOTAL FOR CLEARING STACK, MEMORY
; STACK POINTER NOW $FF, A=X==0
```

Here's the latest source code, with the initialisation inserted, and a few minor changes here and there...

Code:

```
processor 6502
include "vcs.h"
include "macro.h"

;-------------------------------------------------------
SEG
ORG $F000

Reset

; Clear RAM and all TIA registers

ldx #0
lda #0
Clear sta 0,x
inx
bne Clear

StartOfFrame

; Start of vertical blank processing

lda #0
sta VBLANK

lda #2
sta VSYNC

sta WSYNC
sta WSYNC
sta WSYNC ; 3 scanlines of VSYNC signal
```

```
lda #0
sta VSYNC

; 37 scanlines of vertical blank...

ldx #0
VerticalBlank sta WSYNC
inx
cpx #37
bne VerticalBlank


; 192 scanlines of picture...

ldx #0
Picture
SLEEP 20 ; adjust as required!

inx
stx COLUBK

SLEEP 2 ; adjust as required!

txa
eor #$FF
sta COLUBK

sta WSYNC

cpx #192
bne Picture



lda #%01000010
sta VBLANK ; end of screen - enter blanking

; 30 scanlines of overscan...

ldx #0
Overscan sta WSYNC
inx
cpx #30
bne Overscan

jmp StartOfFrame


;--------------------------------------------------------------
ORG $FFFA

InterruptVectors
```

```
.word Reset ; NMI
.word Reset ; RESET
.word Reset ; IRQ

END
```

Session 13: Playfield Basics

In the last few sessions, we started to explore the capabilities of the TIA. We learned that the TIA has "registers" which are mapped to fixed memory addresses, and that the 6502 can control the TIA by writing and/or reading these addresses. In particular, we learned that writing to the WSYNC register halts the 6502 until the TIA starts the next scanline, and that the COLUBK register is used to set the colour of the background. We also learned that the TIA keeps an internal copy of the value written to COLUBK.

Today we're going to have a look at playfield graphics, and for the first time learn how to use RAM. The playfield is quite a complex beast, so we may be spending the next few sessions exploring its capabilities.

The '2600 was originally designed to be more or less a sophisticated programmable PONG-style machine, able to display 2-player games - but still pretty much PONG in style. These typically took place on a screen containing not much more than walls, two "players" - usually just straight lines – and a ball. Despite this, the design of the system was versatile enough that clever programmers have produced a wide variety of games.

The playfield is that part of the display which usually shows "walls" or "backgrounds" (not to be confused with THE background colour). These walls are usually only a single colour (for any given scanline), though games typically change the colour over multiple scanlines to give some very nice effects.

The playfield is also sometimes used to display very large (square, blocky looking) scores and words.

Just like with COLUBK, the TIA has internal memory where it stores exactly 20 bits of playfield data, corresponding to just 20 pixels of playfield. Each one of these pixels can be on (displayed) or off (not displayed).

The horizontal resolution of the playfield is a very-low 40 pixels, divided into two halves - both of which display the same 20 bits held in the TIA internal memory. Each half of the playfield may have its own colour (we'll cover this later), but all pixels either half are the same colour. Each playfield pixel is exactly 4 colour-clocks wide (160 colour clocks / 40 pixels = 4 colour clocks per pixel).

The TIA manages to draw a 40 pixel playfield from only 20 bits of playfield data by duplicating the playfield (the right side of the playfield displays the same data as the left side). It is possible to mirror the right side, and it is also possible to create an "asymmetrical playfield" - where the right and left sides of the playfield are NOT symmetrical. I'll leave you to figure out how to do that for now - we'll cover it in a future session. For now, we're just going to learn how to play with those 20 bits of TIA memory, and see what we can do with them.

Let's get right into it. Here's some sample code which introduces a few new TIA registers, and also (for the first time for us) uses a RAM location to store some temporary information (a variable!). There are three TIA playfield registers (two holding 8 bits of playfield data, and one holding the remaining 4 bits) - PF0, PF1, PF2. Today we're going to focus on just one of these TIA playfield registers, PF1, because it is the simplest to understand.

Code:

```
; '2600 for Newbies
; Session 13 - Playfield


            processor 6502
            include "vcs.h"
            include "macro.h"

;-------------------------------------------------------------------------

PATTERN         = $80               ; storage location (1st byte in RAM)
TIMETOCHANGE    = 20                ; speed of "animation" - change as desired
;-------------------------------------------------------------------------

            SEG
            ORG $F000

Reset

   ; Clear RAM and all TIA registers

            ldx #0
            lda #0
Clear         sta 0,x
            inx
            bne Clear

      ;--------------------------------------------------
      ; Once-only initialisation...

            lda #0
            sta PATTERN             ; The binary PF 'pattern'

            lda #$45
            sta COLUPF              ; set the playfield colour

            ldy #0                  ; "speed" counter

      ;--------------------------------------------------

StartOfFrame

   ; Start of new frame
   ; Start of vertical blank processing

            lda #0
            sta VBLANK

            lda #2
            sta VSYNC

            sta WSYNC
            sta WSYNC
```

```
            sta WSYNC               ; 3 scanlines of VSYNC signal

            lda #0
            sta VSYNC


      ;--------------------------------------------------
      ; 37 scanlines of vertical blank...

            ldx #0
VerticalBlank   sta WSYNC
            inx
            cpx #37
            bne VerticalBlank

      ;--------------------------------------------------
      ; Handle a change in the pattern once every 20 frames
      ; and write the pattern to the PF1 register

            iny                 ; increment speed count by one
            cpy #TIMETOCHANGE       ; has it reached our "change point"?
            bne notyet              ; no, so branch past

            ldy #0              ; reset speed count

            inc PATTERN             ; switch to next pattern
notyet


            lda PATTERN             ; use our saved pattern
            sta PF1             ; as the playfield shape

      ;--------------------------------------------------
      ; Do 192 scanlines of colour-changing (our picture)

            ldx #0              ; this counts our scanline number

Picture       stx COLUBK              ; change background colour (rainbow effect)
            sta WSYNC               ; wait till end of scanline

            inx
            cpx #192
            bne Picture

      ;--------------------------------------------------



            lda #%01000010
            sta VBLANK          ; end of screen - enter blanking

   ; 30 scanlines of overscan...
```

```
            ldx #0
Overscan    sta WSYNC
            inx
            cpx #30
            bne Overscan




            jmp StartOfFrame


;--------------------------------------------------------------------------------

          ORG $FFFA

InterruptVectors

          .word Reset          ; NMI
          .word Reset          ; RESET
          .word Reset          ; IRQ

         END
```

The binary for this code, and a snapshot of the output are included below.

What you will see is our rainbow-coloured background, as before - but over the top of it we see a strange-pattern of vertical stripe(s). And the pattern changes. These vertical stripes are our first introduction to playfield graphics.

Have a good look at what this demo does; although it is only writing to a single playfield register (PF1) which can only hold 8 bits (pixels) of playfield data, you always see the same stripe(s) on the left side of the screen, as on the right. This is a result, as noted earlier, of the TIA displaying its playfield data twice on any scanline - the first 20 bits on the left side, then repeated for the right side.

Let's walk through the code and have a look at some of the new bits...

Code:

```
PATTERN        = $80                ; storage location (1st byte in RAM)
TIMETOCHANGE   = 20                     ; speed of "animation" - change as desired
```

At the beginning of our code we have a couple of equates. Equates are labels with values assigned to them. We have covered this sort of label value assignation when we looked at how DASM resolved symbols when assembling our source code. In this case, we have one symbol (PATTERN) which in the code is used as a storage location

Code:

```
    sta PATTERN
```

... and the other (TIMETOCHANGE) which is used in the code as a number for comparison

Code:

```
   cpy #TIMETOCHANGE
```

Remember how we noted that the assembler simply replaced any symbol it found with the actual value of that symbol. Thus the above two sections of code are exactly identical to writing "sta $80" and "cpy #20". But from our point of view, it's much better to read (and understand) when we use symbols instead of values.

So, at the beginning of our source code (by convention, though you can pretty much define symbols anywhere), we include a section giving values to symbols which are used throughout the code. We have a convenient section we can go back to and "adjust" things later on.

Here's our very first usage of RAM...

Code:

```
            lda #0
            sta PATTERN             ; The binary PF 'pattern'
```

Remember, DASM replaces that symbol with its value. And we've defined the value already as $80. So that "sta" is actually a "sta $80", and if we have a look at our memory map, we see that our RAM is located at addresses $80 - $FF. So this code will load the accumulator with the value 0 (that's what that crosshatch means - load a value, not a load from memory) and then store the accumulator to memory location $80. We use PATTERN to hold the "shape" of the graphics we want to see. It's just a byte, consisting of 8 bits. But as we have seen, the playfield is 20 bits each being on or off, representing a pixel. By writing to PF1 we are actually modifying just 8 of the TIA playfield bits. We could also write to PF0 and PF2 - but let's get our understanding of the basic playfield operation correct, first.

Code:

```
        lda #$45
        sta COLUPF          ; set the playfield colour
```

When we modified the colour of the background, we wrote to COLUBK. As we know, the TIA has its own internal 'state', and we can modify its state by writing to its registers. Just like COLUBK, COLUPF is a colour register. It is used by the TIA for the colour of playfield pixels (which are visible - ie: their corresponding bit in the PF0, PF1, PF2 registers is set).

If you want to know what colour $45 is, look it up in the colour charts presented earlier. I just chose a random value, which looks reddish to me

Code:

```
        ldy #0              ; "speed" counter
```

We should be familiar with the X,Y and A registers by now. This is loading the value 0 into the y register. Since Y was previously unused in our kernel, for this example I am using it as a sort of speed throttle. It is incremented by one every frame, and every time it gets to 20 (or more precisely, the value of TIMETOCHANGE) then we change the pattern that is being placed into the PF1 register. We change the speed at which the pattern changes by changing the value of the TIMETOCHANGE equate at the top of the file.

That speed throttle and pattern change is handled in this section...

Code:

```
    ; Handle a change in the pattern once every 20 frames
    ; and write the pattern to the PF1 register

        iny             ; increment speed count by one
        cpy #TIMETOCHANGE   ; has it reached our "change point"?
        bne notyet          ; no, so branch past

        ldy #0          ; reset speed count

        inc PATTERN         ; switch to next pattern
notyet


        lda PATTERN         ; use our saved pattern
        sta PF1         ; as the playfield shape
```

This is the first time we've seen an instruction like "inc PATTERN" - the others we have already covered. "inc" is an increment - and it simply adds 1 to the contents of any memory (mostly RAM) location. We initialised PATTERN (which lives at $80, remember!) to 0. So after 20 frames, we will find that the value gets incremented to 1. 20 frames after that, it is incremented to 2.

Now let's go back to our binary number system for a few minutes. Here's the binary representation of the numbers 1 to 10

00000000
00000001
00000010
00000011
00000100
00000101
00000111
00001000
00001010

Have a real close look at the pattern there, and then run the binary again and look at the pattern of the stripe. I'm telling you, they're identical! That is because, of course, we are writing these values to the PF1 register and where there is a set bit (value of 1) that corresponds directly to a pixel being displayed on the screen.

See how the PF1 write is outside the 192-line picture loop. We only ever write the PF1 once per frame (though we could write it every scanline if we wished). This demonstrates that the TIA has kept the value we write to its register(s) and uses that same value again and again until it is changed by us.

The rest of the code is identical to our earlier tutorials - so to get our playfield graphics working, all we've had to do is write a colour to the playfield colour register (COLUPF), and then write actual pixel data to the playfield register(s) PF0, PF1 and PF2. We've only touched PF1 this time - feel free to have a play and see what happens when you write the others.

You might also like to play with writing values INSIDE the picture (192-line) loop, and see what happens when you play around with the registers 'on-the-fly'. In fact, since the TIA retains and redraws the same thing again and again, to achieve different 'shapes' on the screen, this is exactly what we have to do - write different values to PF0, PF1, PF2 not only every scanline, but also change the shapes in the middle of a scanline!

Today's session is meant to be an introduction to playfield graphics - don't worry too much about the missing information, or understanding exactly what's happening. Try and have a play with the code, do the exercises - and next session we should have a more comprehensive treatment of the whole shebang.

==========
Exercises

1. Modify the kernel so that instead of showing a rainbow-colour for the background, it is the playfield which has the rainbow effect.

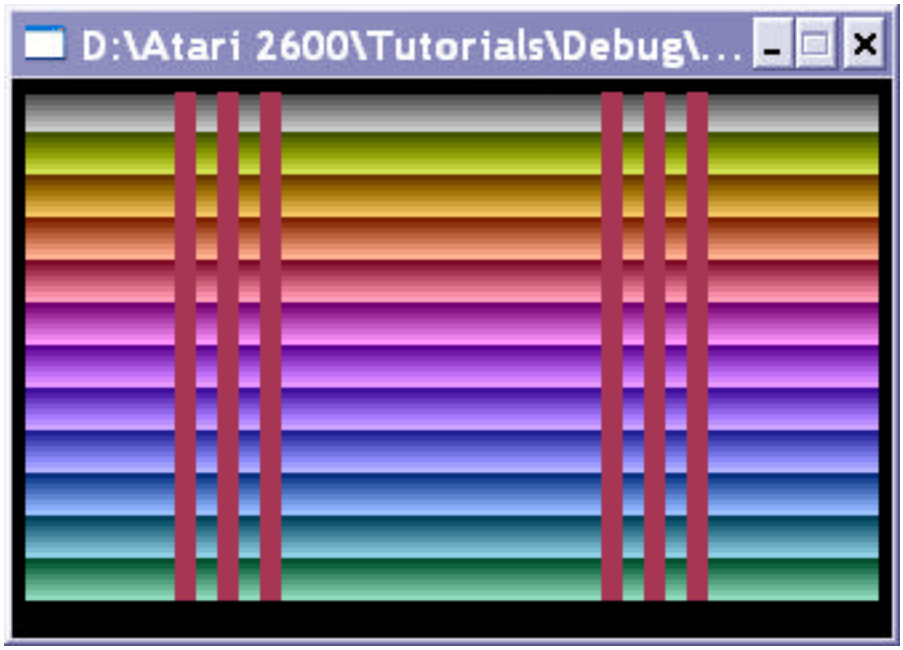
2. What happens when you use PF0 or PF2 instead of PF1? It can get pretty bizarre - we'll explain what's going on in the next session.

3. Can you change the kernel so it only shows *ONE* copy of the playfield you write (that is, on the left side you see the pattern, and on the right side it's blank). Hint: You'll need to modify PF1 mid-scanline.
We'll have a look at these exercises next session. Don't worry if you can't understand or implement them - they're pretty tricky.

==========

Subjects we will tackle next time include...

* The other playfield registers (PF0, PF2)
* The super-weird TIA pixel -> screen pixel mapping
* Mirrored playfields
* Two colours playfields
* Asymmetrical playfield



Errata

Rather than edit my original messages, and hope nobody notices... I've decided to point out my original errors in this errata, so that people are aware what I originally said - and what I should have said.

In Session 1 (the first post), I talked about EEPROM. These are electrically-erasable Read-Only Memory, and I never used them. I should have said EPROM - which are erased with ultra-violet light.

In Session 2 I mention SECAM being used in France and Russia. SECAM is also used in the middle east and other ex-French colonies (Viet Nam) SECAM is very similar to PAL (625/50Hz).

Also in session 2, I wrote

"The Atari 2600 *ONLY* sends the TV the "colour and intensity information for the electron beam as it sweeps across that line". The '2600 programmer needs to feed the TV the rest of the information - basically, the signal to start the image frame, and the signal to tell it to start the next scanline, and the next.... etc."

This is not correct. Every scanline, the TIA automatically inserts the correct signal for the start of the scanline (it's called the HSYNC signal). The vertical synchronisation signal (VSYNCH) which tells the TV when to start a new frame is the responsibility of the programmer.

In Session 3, I wrote...

"A side-note: 76 cycles per line x 262 lines per frame x 50 frames per second = the number of 6502 cycles per second for NTSC (= 1.19MHz, roughly)."

Astute readers will have noticed that NTSC is 60 frames per second, not 50.

Thanks to Eric Ball for his suggestions and corrections.