Version 1.0

"Inside the XGS AVR 8-Bit"

User Manual and Programming Guide

Andre' LaMothe

Nurve Networks LLC

Author
Andre' LaMothe

Editor/Technical Reviewer
The "Collective"

Printing
0001

ISBN
Pending

Licensing, Terms & Conditions

NURVE NETWORKS LLC, . END-USER LICENSE AGREEMENT FOR XGS AVR HARDWARE, SOFTWARE , EBOOKS, AND USER MANUALS

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS PRODUCT. IT CONTAINS SOFTWARE, THE USE OF WHICH IS LICENSED BY NURVE NETWORKS LLC, INC., TO ITS CUSTOMERS FOR THEIR USE ONLY AS SET FORTH BELOW. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT USE THE SOFTWARE OR HARDWARE. USING ANY PART OF THE SOFTWARE OR HARDWARE INDICATES THAT YOU ACCEPT THESE TERMS.

GRANT OF LICENSE: NURVE NETWORKS LLC (the "Licensor") grants to you this personal, limited, non-exclusive, non-transferable, non-assignable license solely to use in a single copy of the Licensed Works on a single computer for use by a single concurrent user only, and solely provided that you adhere to all of the terms and conditions of this Agreement. The foregoing is an express limited use license and not an assignment, sale, or other transfer of the Licensed Works or any Intellectual Property Rights of Licensor.

ASSENT: By opening the files and or packaging containing this software and or hardware, you agree that this Agreement is a legally binding and valid contract, agree to abide by the intellectual property laws and all of the terms and conditions of this Agreement, and further agree to take all necessary steps to ensure that the terms and conditions of this Agreement are not violated by any person or entity under your control or in your service.

OWNERSHIP OF SOFTWARE AND HARDWARE: The Licensor and/or its affiliates or subsidiaries own certain rights that may exist from time to time in this or any other jurisdiction, whether foreign or domestic, under patent law, copyright law, publicity rights law, moral rights law, trade secret law, trademark law, unfair competition law or other similar protections, regardless of whether or not such rights or protections are registered or perfected (the "Intellectual Property Rights"), in the computer software and hardware, together with any related documentation (including design, systems and user) and other materials for use in connection with such computer software and hardware in this package (collectively, the "Licensed Works"). ALL INTELLECTUAL PROPERTY RIGHTS IN AND TO THE LICENSED WORKS ARE AND SHALL REMAIN IN LICENSOR.

RESTRICTIONS:
(a) You are expressly prohibited from copying, modifying, merging, selling, leasing, redistributing, assigning, or transferring in any matter, Licensed Works or any portion thereof.
(b) You may make a single copy of software materials within the package or otherwise related to Licensed Works only as required for backup purposes.
(c) You are also expressly prohibited from reverse engineering, decompiling, translating, disassembling, deciphering, decrypting, or otherwise attempting to discover the source code of the Licensed Works as the Licensed Works contain proprietary material of Licensor. You may not otherwise modify, alter, adapt, port, or merge the Licensed Works.
(d) You may not remove, alter, deface, overprint or otherwise obscure Licensor patent, trademark, service mark or copyright notices.
(e) You agree that the Licensed Works will not be shipped, transferred or exported into any other country, or used in any manner prohibited by any government agency or any export laws, restrictions or regulations.
(f) You may not publish or distribute in any form of electronic or printed communication the materials within or otherwise related to Licensed Works, including but not limited to the object code, documentation, help files, examples, and benchmarks.

TERM: This Agreement is effective until terminated. You may terminate this Agreement at any time by uninstalling the Licensed Works and destroying all copies of the Licensed Works both HARDWARE and SOFTWARE. Upon any termination, you agree to uninstall the Licensed Works and return or destroy all copies of the Licensed Works, any accompanying documentation, and all other associated materials.

WARRANTIES AND DISCLAIMER: EXCEPT AS EXPRESSLY PROVIDED OTHERWISE IN A WRITTEN AGREEMENT BETWEEN LICENSOR AND YOU, THE LICENSED WORKS ARE NOW PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THE WARRANTY OF NON-INFRINGEMENT. WITHOUT LIMITING THE FOREGOING, LICENSOR MAKES NO WARRANTY THAT (i) THE LICENSED WORKS WILL MEET YOUR REQUIREMENTS, (ii) THE USE OF THE LICENSED WORKS WILL BE UNINTERRUPTED, TIMELY, SECURE, OR ERROR-FREE, (iii) THE RESULTS THAT MAY BE OBTAINED FROM THE USE OF THE LICENSED WORKS WILL BE ACCURATE OR RELIABLE, (iv) THE QUALITY OF THE LICENSED WORKS WILL MEET YOUR EXPECTATIONS, (v) ANY ERRORS IN THE LICENSED WORKS WILL BE CORRECTED, AND/OR (vi) YOU MAY USE, PRACTICE, EXECUTE, OR ACCESS THE LICENSED WORKS WITHOUT VIOLATING

THE INTELLECTUAL PROPERTY RIGHTS OF OTHERS. SOME STATES OR JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY MAY LAST, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IF CALIFORNIA LAW IS NOT HELD TO APPLY TO THIS AGREEMENT FOR ANY REASON, THEN IN JURISDICTIONS WHERE WARRANTIES, GUARANTEES, REPRESENTATIONS, AND/OR CONDITIONS OF ANY TYPE MAY NOT BE DISCLAIMED, ANY SUCH WARRANTY, GUARANTEE, REPRESENATION AND/OR WARRANTY IS: (1) HEREBY LIMITED TO THE PERIOD OF EITHER (A) Five (5) DAYS FROM THE DATE OF OPENING THE PACKAGE CONTAINING THE LICENSED WORKS OR (B) THE SHORTEST PERIOD ALLOWED BY LAW IN THE APPLICABLE JURISDICTION IF A FIVE (5) DAY LIMITATION WOULD BE UNENFORCEABLE; AND (2) LICENSOR'S SOLE LIABILITY FOR ANY BREACH OF ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION, AND/OR CONDITION SHALL BE TO PROVIDE YOU WITH A NEW COPY OF THE LICENSED WORKS. IN NO EVENT SHALL LICENSOR OR ITS SUPPLIERS BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, THOSE RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT LICENSOR HAD BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE LICENSED WORKS. SOME JURISDICTIONS PROHIBIT THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

SEVERABILITY: In the event any provision of this License Agreement is found to be invalid, illegal or unenforceable, the validity, legality and enforceability of any of the remaining provisions shall not in any way be affected or impaired and a valid, legal and enforceable provision of similar intent and economic impact shall be substituted therefore.

ENTIRE AGREEMENT: This License Agreement sets forth the entire understanding and agreement between you and NURVE NETWORKS LLC, supersedes all prior agreements, whether written or oral, with respect to the Software, and may be amended only in a writing signed by both parties.

NURVE NETWORKS LLC
12724 Rush Creek Lane
Austin, TX 78732
support@nurve.net
www.xgamestation.com

Version & Support/Web Site

This document is valid with the following hardware, software and firmware versions:

- XGS AVR 8-Bit Game Console Revision A. or greater.
- Atmel AVR Studio 4.14 or greater.

The information herein will usually apply to newer versions but may not apply to older versions. Please contact Nurve Networks LLC for any questions you may have.

Visit **www.xgamestation.com** for downloads, support and access to the XGameStation user community and more!

For technical support, sales, general questions, share feedback, please contact Nurve Networks LLC at:

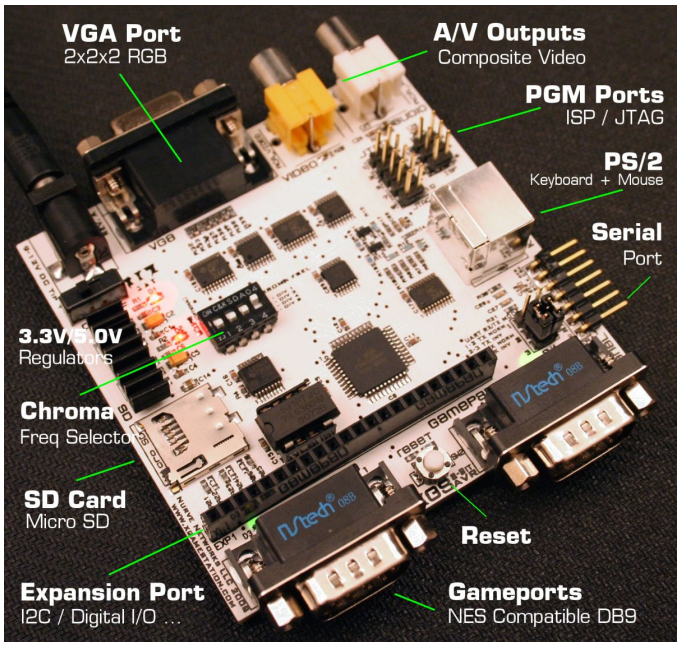
support@nurve.net / **nurve_help@yahoo.com**

Inside the XGS AVR 8-Bit

User Manual and Programming Guide (SAMPLE)

1.0 XGS AVR 8-Bit Overview (SAMPLE)

Figure 1.1 – The XGS AVR 8-Bit.



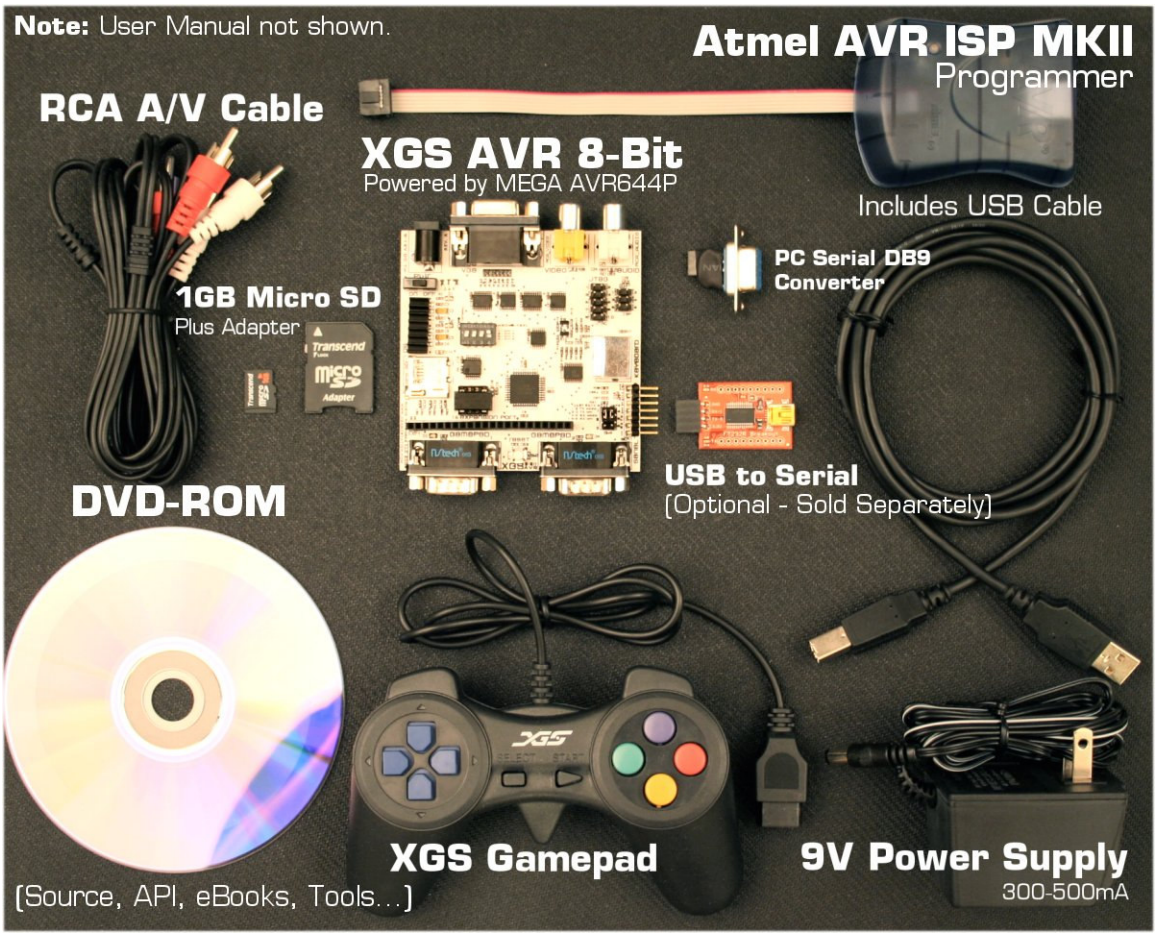
The **XGS AVR 8-Bit** or simply the **"XGS AVR"** is developed around the **Atmel 8-Bit Mega AVR 644 QFP** (more specifically the Pico Power Version 644P). Figure 1.1 shows an image of the XGS AVR 8-bit annotated. The XGS AVR has the following hardware features:

- 44-Pin QFP Package of the AVR 644, runs 20 MHz, but we are over clocking to 28.636360 MHz (8 X NTSC).
- RCA Video and Audio Out Ports.
- HD15 Standard VGA Out Port.
- Single PS/2 Keyboard (and mouse) Port.
- Two DB9 Nintendo NES/Famicom compatible gamepad ports (using Custom XGS Controllers).
- Micro SD card interface.
- Single 9V DC power in with regulated output of 5.0V @ 500mA and 3.3V @ 500 mA on board to support external peripherals and components (the AVR 644 is 5V).
- Removable XTAL oscillator to support faster speeds and experimenting with various reference clocks.
- Game/Expansion Port that exposes I/O, power, clocking ,etc.
- Standard RS-232 serial port brought out with 7-pin header interface for "hobbyists" as well as standard available USB to serial converters manufactured by Parallax (USB 2 SER) and Sparkfun.
- 6-PIN ISP (**In System Programming**) interface compatible with Atmel **AVRISP MKII** as well as other 3rd party programmers.
- 10-Pin JTAG interface for programmers and debuggers such as the Atmel **AVR JTAG ICE MKII** as well as other 3rd party programmers.

The XGS AVR is more or less a minimal platform to show off the capabilities of the AVR 644 and have fun learning it. Next, let's inventory your XGS AVR 8-Bit Kit.

1.1 Package Contents

Figure 1.2 – XGS AVR kit contents.

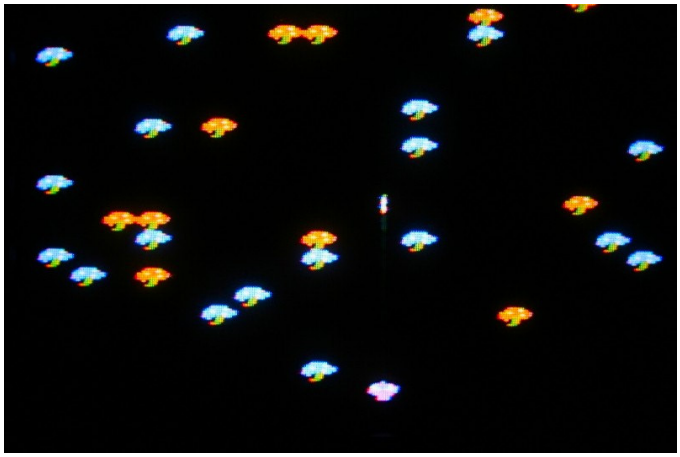


You should have the following items in your kit, referring to Figure 1.2.

- XGS AVR 8-Bit Game Console.
- 71 GB blank micro SD card and standard size SD adapter.
- Standard USB cable for ISP programmer.
- 9V 500 mA, DC unregulated wall adapter with 2.1 plug and tip (+) positive, ring (-) negative.
- RCA A/V cable.
- XGS Nintendo internal controller with custom DB9 connector on it.
- XGS DB9 Serial Adapter dongle (connects PC DB9 serial to XGS serial header).
- DVD ROM with all the software, demos, IDE, tools, and this document.
- Atmel ISP MKII programmer.
- User Manual (not shown).

1.2 XGS AVR "Quick Start" Demo

Figure 1.3 – The NTSC demo running.



The XGA AVR 8-bit is pre-loaded with a simple tile mode demo programmed into the AVR's FLASH memory, a screen shot is shown in Figure 1.3. We will use this to test your system out. The following are a series of steps to try the demo out and make sure your hardware is working.

Step 1: Place your XGS AVR on a flat surface, no carpet! Static electricity!

Step 2: Make sure the power switch at the front is in the **OFF** position, this is to the **LEFT**.

Step 3: Plug your wall adapter in and plug the 2.1mm connector into the female port located top-left corner of the XGS AVR.

Step 4: Insert the A/V cable into the yellow (video) and white (audio) port of the XGS AVR located top-right of the board and then insert them into your NTSC/Multi-System TV's A/V port.

Step 5: Plug the XGS game controller into the **LEFT** controller port on the XGS.

Step 6: Turn the power on by sliding the ON/OFF switch to the **RIGHT**.

Step 7: The demo will start immediately, it's a skeleton of a Centipede game. Use the gamepad's Dpad to move the player and the Action 2 button (yellow) to fire.

You should see something like that shown in Figure 1.3. The actual program that is loaded into the AVR is located on your DVD here:

DVD-ROM:\ XGSAVR \ SOURCES \ XGS_CENTIPEDE_NTSC_01.C

Of course, it needs many other driver and system files to link with, but we will get to this latter when we discuss the installation of AVR Studio/WinAVR and the tool chain in general for C/C++ and ASM programming.

The demo is nothing more than a simple tile engine demo of one of the many NTSC tiles engines, this tile engine happens to support 20x24 tiles, 8x8 bitmaps for the tiles with 8-bit color per pixel, along with course horizontal and vertical scrolling along with fine vertical scrolling support. The controls are shown in Table 1.1.

Table 1.1 – Tile demo game pad controls.

Action	Control
Fire	Action 2
Right	DPad Right
Left	DPad Left
Up	DPad Up
Down	DPad Down

Hit the **Reset** button over and over and the demo will reset and reload immediately, If the system ever locks up (rare, and always due to bad code), then simply hit **Reset** a few times or cycle the power. This concludes the ***Quick Start*** demo.

Figure 1.4 – The Atmel Mega AVR 644/P packing for 40-Pin DIP and the 44-Pin QFP.

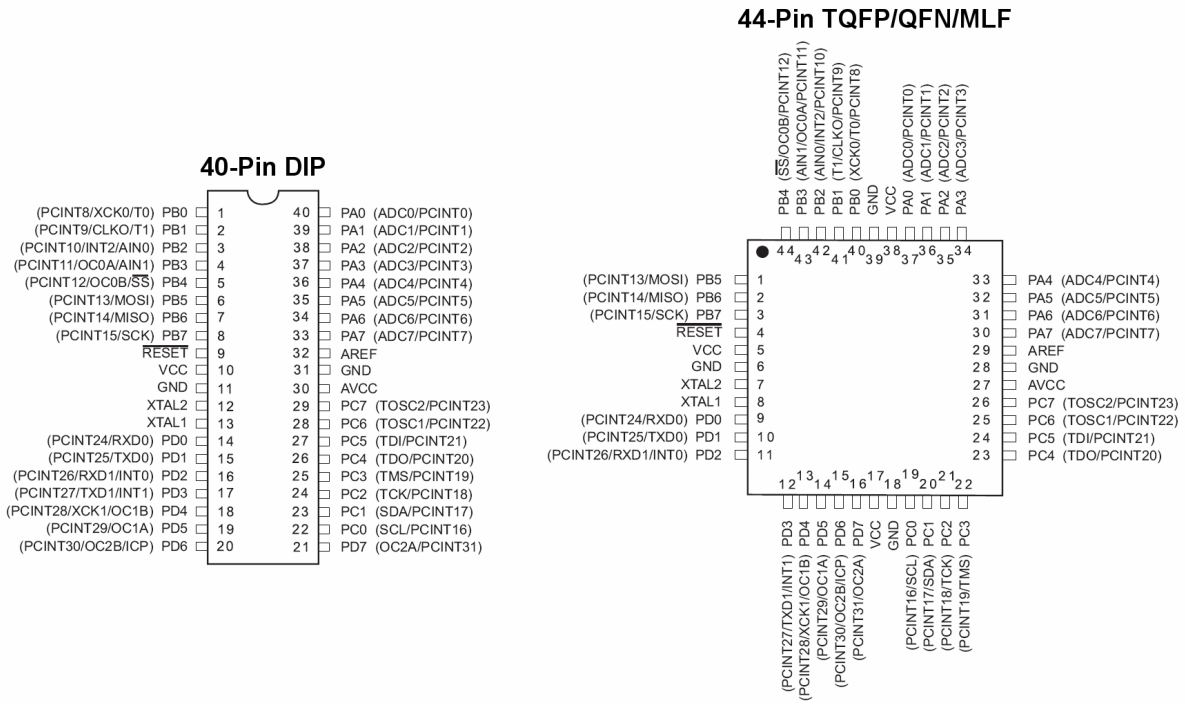
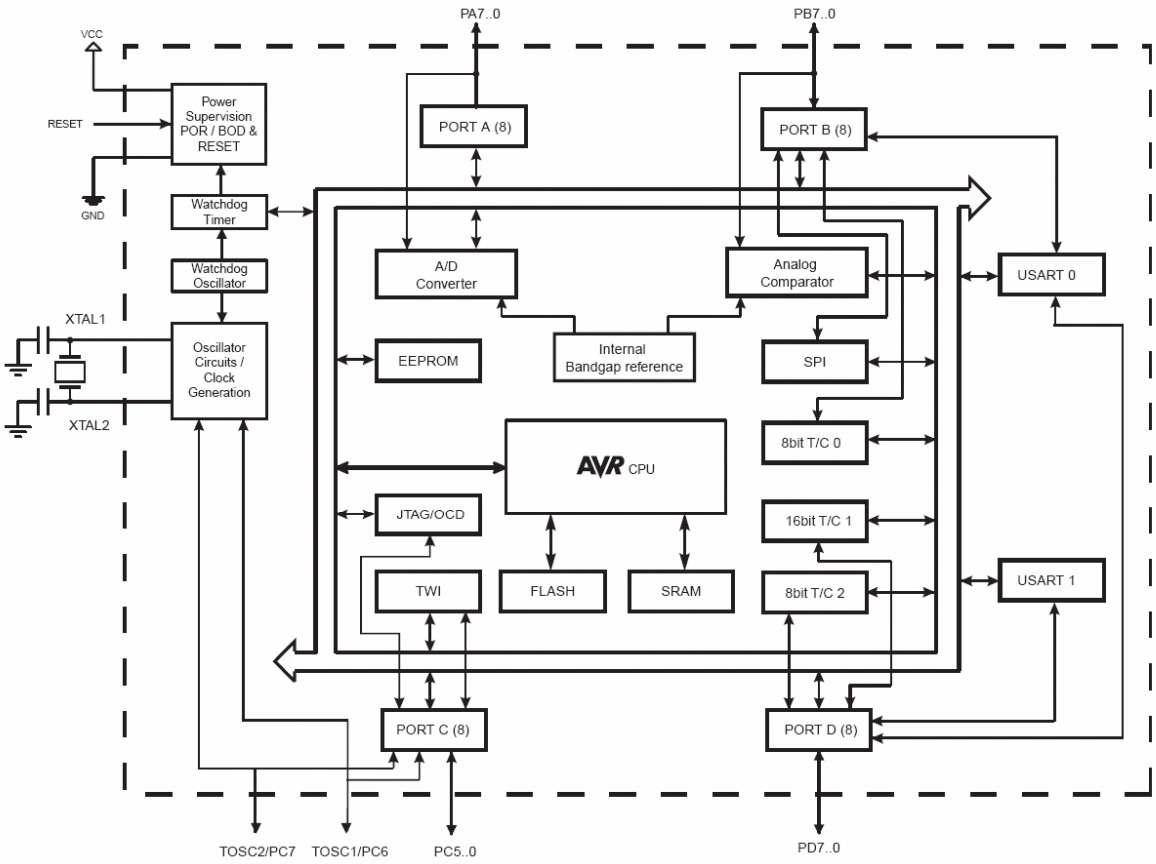


Figure 1.5 – The Atmel Mega AVR 644/P architecture block diagram.



The AVR core combines a rich instruction set with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

1.3 The Atmel Mega AVR 644/P Chip

The Atmel AVR 644P (P stands for Pico Power) comes in a number of packages including 40-pin **DIP** (duel inline package) and 44-pin **QFP** (quad flat pack) type package. We are using the Mega AVR 644 QFP package on the XGS. Figure 1.4 shows the packaging and pin out of the chip. The AVR 644 was designed as a general purpose, high performance microcontroller with an 8-bit data path, single clock execution on many instructions, as well as support for 16-bit operations and built in multiply instructions. The chip has a large FLASH memory of **64K** Bytes, but a smaller **4K** Byte SRAM which makes a lot of graphical applications challenging, but that's half the fun of developing applications on the AVR 8-bit processors and why we are all here! Nonetheless, the idea of the XGS AVR isn't just for graphics and sound, but for you to learn the AVR line of chips, specifically the MEGA AVRs in a fun way using graphics and sound as the platform for engagement.

Also, Table 1.2 below shows the various differences between the 164, 324, and 644 variants for reference. Additionally, there is a useful EEPROM memory as well that can be used as a little disk drive or memory storage for whatever purposes. Of course, the AVR 644 allows re-writing to the FLASH as well, so unused portions of FLASH can be used for storage; however, it's not ideal to constantly re-write FLASH since there is a limit to the number of times it can be re-written; 100,000 give or take.

NOTE	FLASH memories typically have a maximum number of times they can be written; something in the range of 10,000 to 100,000. This doesn't mean that at 10,001 or 100,001 the memory won't work, it just means the erase cycles and write cycles may take longer to get the memory to clear or write. And this then degrades further as the write/erase cycles persist. Thus, if you were to code all day and re-write your FLASH 100x times a day, then at 100,000 re-write cycles, you would have 3-4 years before you ever saw any problems. On the other hand, if you write code to use the FLASH as a solid state disk and constantly re-write the memory 10,000x a run, you can see how quickly you might degrade the memory. Thus, use the EEPROM for memory you need to update and still be non-volatile and save the life of the FLASH.

Table 1.2 - Differences between ATmega164P, 324P, and 644P.

Device	Flash	EEPROM	RAM
ATmega164P	16 Kbyte	512 Bytes	1 Kbyte
ATmega324P	32 Kbyte	1 Kbyte	2 Kbyte
ATmega644P	64 Kbyte	2 Kbyte	4 Kbyte

Note: *The "P" suffix simply means "Pico Power" and has nothing to do with the chip operation or functionality. The Pico power version is identical to the non-pico power for our purposes and I will use them interchangeably.*

Figure 1.5 shows the AVR 644 architecture in block diagram form and Table 1.3 lists the pins and their function for the AVR 644.

NOTE	Since the AVR 644 has so many internal peripherals and only a finite number of pins, many functions are multiplexed on the I/O pins such as SPI, I2C, UARTs, A/D, D/A, etc. Thus, when you enable one of the peripherals they will typically override the I/O functionality and take on the special functions requested. However, when you don't enable any peripherals each I/O pin is a simple I/O pin as listed in Table 1.3.

Table 1.3 – The AVR 644 general pin descriptions.

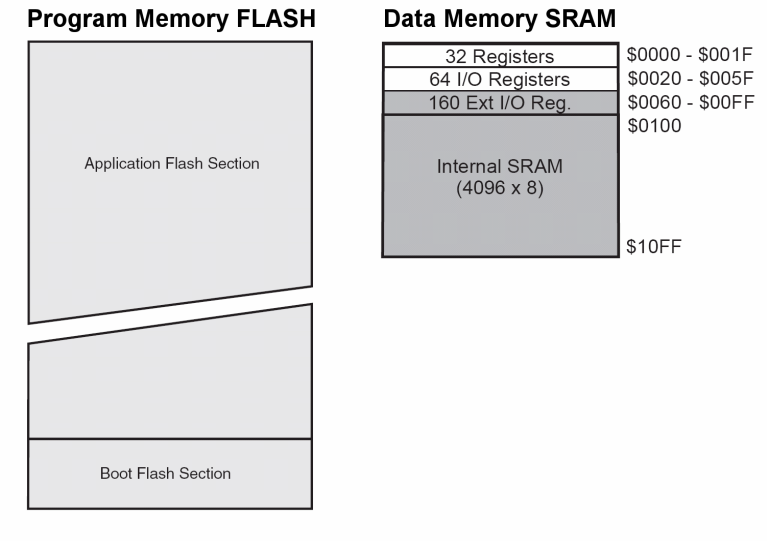
Pin Group	Description
Port A (PA7:PA0) - Port A serves as analog inputs to the Analog-to-digital Converter. Port A also serves as an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port A output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port A pins that are externally pulled low will source current if the pull-up resistors are activated. The Port A pins are tri-stated when a reset condition becomes active, even if the clock is not running. Port A also serves the functions of various special features of the ATmega644P.	
Port B (PB7:PB0) - Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.	
Port C (PC7:PC0) - Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running. Port C also serves the functions of the JTAG interface, along with special features of the ATmega 644P.	

Port D (PD7:PD0) - Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running. Port D also serves the functions of various special features of the ATmega 644P.
/**RESET –** Reset input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running.
XTAL1 - Input to the inverting Oscillator amplifier and input to the internal clock operating circuit.
XTAL2 – Output from the inverting Oscillator amplifier.
AVCC - AVCC is the supply voltage pin for Port F and the Analog-to-digital Converter. It should be externally connected to VCC, even if the ADC is not used. If the ADC is used, it should be connected to VCC through a low-pass filter.
AREF - This is the analog reference pin for the Analog-to-digital Converter.

The AVR 644 is a ***8-bit RISC-like*** architecture chip with instructions being either ***16 or 32-bits*** in size (mostly 16-bit). The memory model is a ***"Hardware Architecture"*** meaning that the data and memory are located in separate memories that are not addressed as a contiguous space, but rather as separate memories with different instructions to read/write to them. This allows faster execution since the same buses aren't used to access data and program space. Therefore, you will typically access SRAM as a continuous 4K of memory and program/FLASH memory is in a completely different address space as is EEPROM memory. Thus, there are 3 different memories that the AVR 644 supports. Additionally, the AVR 644 maps registers as well as all it's I/O ports in the SRAM memory space for ease of access. Figure 1.6 show these memories.

TIP	***Harvard*** as opposed to ***Von Neumann*** architecture are the two primary computer memory organizations used in modern processors. Harvard was created at Harvard University, thus the moniker, and likewise Von Neumann was designed by mathematician John Von Neumann Von Neumann differs from Harvard in that Von Neumann uses a single memory for both data and program storage.

Figure 1.6 – FLASH and SRAM memory layouts.



Referring to Figure 1.6, ***Program Memory*** is composed of both a ***"Boot FLASH Section"*** and the ***"Application Section"***. The boot section holds boot ROM code and can be different sizes or disabled. The application section then holds the actual run-time code for the application. The Data Memory is stored in SRAM of course and is **4096** bytes in length. However, due to register space allocation there is a shift in the addresses; the first 256 byte addresses are used to access registers and system I/O then from address **[$0100 - $10FF]** is the 4K block of free memory. Not shown in the memory map is the **"stack"** which will need to go somewhere when you run your code, the C/C++ compiler sets this up for you. When programming in pure ASM, you would have to set the stack pointer appropriately. Speaking of assembly language, here's a little snippet of code excerpted from one of the graphics engines to give you an idea what AVR ASM looks like:

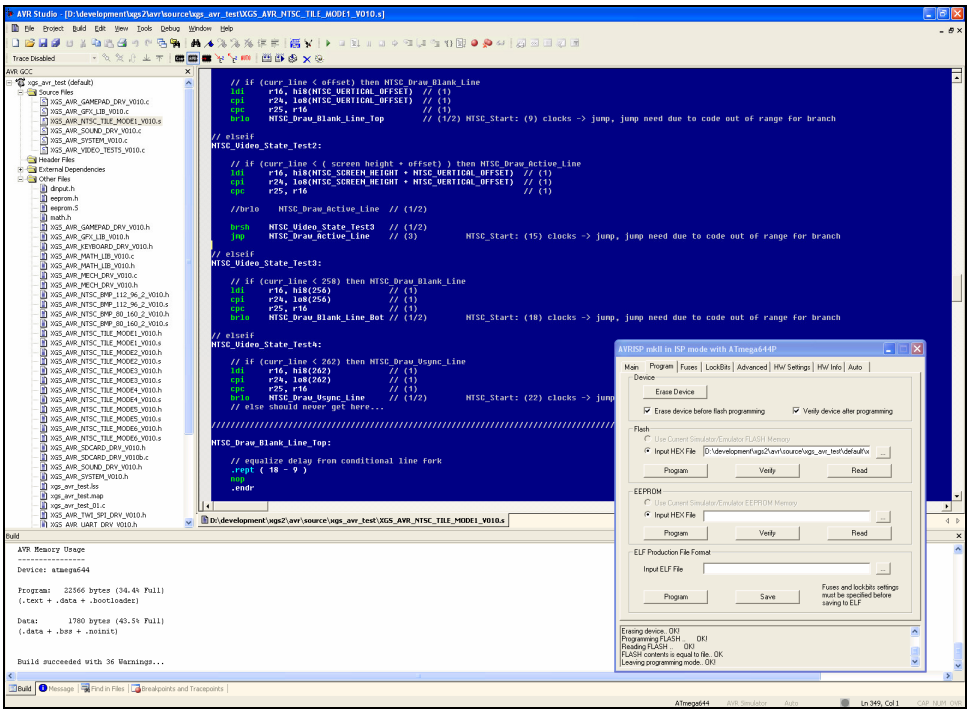
```
NTSC_Reset_Kernel:

    // reset all data structures each frame at line 0

    // initialize any variables for algorithm

    // initialize raster_line for next frame, take into consideration vertical scroll value
    lds     r16, tile_map_vscroll           // (2)
    sts     curr_raster_line, r16           // (2)

    // 16 color clocks sync (128 clocks),
    // this needs to be adjusted correctly to take above computation into consideration..
    DELAYX r16, (18*CLOCKS_PER_PIXEL - 12)

    // disable hsync, maintain black
    ldi     r16, NTSC_BLACK                 // (1)
    out     PORTA, r16                      // (1)

    // now color burst
    DELAYX r16, (2*CLOCKS_PER_PIXEL)

    ldi     r16, NTSC_CBURST0               // (1)
    out     PORTA, r16                      // (1)

    // now color burst
    DELAYX r16, (10*CLOCKS_PER_PIXEL)

    // back to black HSYNC off
    ldi     r16, NTSC_BLACK                 // (1)
    out     PORTA, r16                      // (1)

    jmp     NTSC_Next_Line                  // (3)
```

The snippet shows off a few instructions along with the macro assembler support. The numbers to the right are how many clock cycles each instruction takes -- a good idea when writing ASM code, so you can track the timing if you are writing graphics drivers or any other time critical code.

The AVR C/C++ compiler is based on the ***GNU GCC*** tool chain and thus is not the best optimizing compiler on the planet, but isn't bad. Hence, our approach will be to use C primarily for our coding and make calls to ASM drivers for graphics, sound, etc. Then for ultra high performance applications and games you should use 100% ASM. But, we want to rely on C as much as possible for ease of use. ASM should be used for drivers when necessary via APIs that can be called from C.

NOTE	The tool of choice for AVR development is of course ***AVR Studio***. This tool was developed by Atmel and supports source level debugging, various programmer and ICE debuggers etc. However, the C/C++ compiler is a plug in based on GNU GCC called ***"WinAVR"***. We will discuss the installation of the tool chain shortly, but keep in mind the separation. Additionally, we will be using straight "C" for coding. C++ is supported, but due to its overhead and lack of 100% support for embedded applications, we will avoid it. C is much more compatible with all embedded systems and C++ is just asking for trouble especially with the compiler.

Figure 1.7 – Atmel AVR Studio 4 in action.



AVR Studio is shown in Figure 1.7, it's a standard Windows application that is used to develop applications for the entire line of AVR microcontrollers. It's very similar to Visual C++, but not as polished, you will find little quirks with file management focus issues, etc. The IDE allows viewing of the various files and statistics as builds are performed. The IDE supports the built in Atmel assembler directly or GNU GCC. When in Atmel ASM mode then all your programs must be ASM files using Atmel ASM directives and syntax, they are assembled and linked and a final binary image is generated ready for download to the target (XGS in this case). This is ***not*** how we will use the tool in most cases. Typically, we will use it in GCC mode, so we can code in both C and ASM. However, when you use the GCC mode of operation you are using the entire GCC tool chain including GCC (the C compiler), GAS (GNU assembler), make, librarian and so forth. Luckily, all of this is handled for you automatically as are the creation of Make scripts and so forth.

But, when you use GCC and the GNU assembler GAS, you must use all GNU syntax which is fine for C, but a bit tedious for ASM. The Atmel assembler is a little cleaner in my opinion, but GAS is fine as well. In any event, when you create a new project for the XGS you will select GCC and then you can write .C or .S (asm) files and then will be compiled, assembled and linked together. More of the mechanics of this later, but that's the idea.

Additionally, you can use inline assembly with GCC, but the syntax is terrible as shown in the code snippet below which simply swaps a pair of integers:

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
             "mov %A0, %D0"         "\n\t"
             "mov %D0, __tmp_reg__" "\n\t"
             "mov __tmp_reg__, %B0" "\n\t"
             "mov %B0, %C0"         "\n\t"
             "mov %C0, __tmp_reg__" "\n\t"
             : "=r" (value)
             : "0" (value)
            );
```

As you can see there is a lot of pontification with syntax. If you are a 80xxx coder and have used the VC++ inline assembler or Borland for that matter you should be appalled at the above syntax.

Hence, I personally avoid the inline assembler, but if you want to write some inline ASM you can. However, I recommend against this since it makes it hard to port. Better, to simply add an external .S ASM file and then put your ASM functions in there, and call them from C. This way, you use straight C, and straight ASM, and both are easily portable to other compilers and assemblers by other vendors. Anyone that has used the GNU GCC inline assemblers knows they would rather poke dull forks in their eyes!

1.3.1 System Startup and Reset Details

The XGS AVR has no operating system or external peripherals that need initialization, thus the AVR 644 more or less is the entire "system". When the AVR powers up out of reset, it will either load the boot loader (if there is one), or begin execution from the primary application memory of the FLASH. The initial configuration of the chip is defined by the "fuse" bit settings which are controlled by the programming of the chip. In our case, the fuse settings are mostly default, no boot loader, and run from an external high speed clock source. We will discuss exact details when we get to the software and IDE configuration. Once the AVR 644 boots and starts executing code, whatever is in the application program area is executed, simple as that. With that in mind, now let's discuss every one of the hardware modules in the design.

End Sample

16.0 Graphics Library Module Primer (SAMPLE)

The **"Graphics"** module is the most complex module with the most source files. The graphics generation on the XGS AVR are of course completely ***"software generated"*** by controlling the NTSC/PAL or VGA signal manually using very high speed assembly language and critical timing algorithms. In other words, there is no graphics acceleration or chip. This is by design. One of the main ideas of the XGS AVR (and XGS PIC) is to show how these microcontrollers can be used to perform tasks that are thought to be extremely difficult, impossible, or impractical.

Considering this, there is a lot to talk about in relation to the libraries and how all the pieces fit together. In the following sections we will discuss a number concepts as well as specifically touch on each and every driver and what it does and how it works. Following all of which will be the actual API listings. Here's what's in store:

- Graphics Drivers and System Level Architecture.
- Bitmap Graphics Primer.
- Tile Mapped Graphics Primer.
- The GFX Main Graphics Source Module.
- Putting it all Together: Building Graphics Applications.

Also, note that computer graphics is a huge subject that we can't teach completely in this little manual, this the following information and examples are to get you started. The "demos" that follow later in the manual give concrete examples of using the graphics drivers themselves, so if something doesn't make sense or I leave out some detail, you will probably be able to pick it up from the example demos. Thus, the sections below are more or less an overview of the graphics drivers along with a primer on bitmap and tile graphics, but due to the huge amount of information they can't cover everything. To that end, if you are really interested in computer graphics and game development then make sure to check out the two free eBooks I have provided on the DVD:

- **"Tricks of the Windows Game Programming Gurus"**, 1st Edition.
- **"The Black Art of 3D Game Programming"**, 1st Edition.

They are geared toward PC game programming DirectX, and DOS, but this isn't important really. The graphics programming, algorithms, coverage of bitmapped, and tile graphics are perfectly applicable to the XGS AVR. And hey, they are free, so who can complain! They are located here:

DVD-ROM:\ XGSAVR \ DOCS \ eBooks

16.1 Graphics Drivers and System Level Architecture

Before we begin, Table 16.1 lists the graphics library source files out one again for reference, so you can see how much there is to talk about.

Table 16.1 – Graphics library and driver files.

Library Module Name	Description
High Level Graphics Modules	
XGS_AVR_GFX_LIB_V010.c|h	Graphics initialization, setup, and bitmap rendering primarily.
NTSC Bitmap and Tile Engines	
Bitmap Engines ►	
XGS_AVR_NTSC_BMP_112_96_2_V010.s|h	112x96 2-bits per pixel (4 color) NTSC graphics driver.
XGS_AVR_NTSC_BMP_80_160_2_V010.s|h	80x160 2-bits per pixel (4 color) NTSC graphics driver.
Tile Engines ►	

XGS_AVR_NTSC_TILE_MODE1_V010.s|h	160x192, 20x24, 8x8 tiles resolution NTSC driver tile engine.
XGS_AVR_NTSC_TILE_MODE2_V010.s|h	180x192, 22x24, 8x8 tiles resolution NTSC driver tile engine.
XGS_AVR_NTSC_TILE_MODE3_V010.s|h	208x208, 26x26, 8x8 tiles resolution NTSC driver tile engine.
XGS_AVR_NTSC_TILE_MODE4_V010.s|h	240x208, 30x26, 8x8 tiles resolution NTSC driver tile engine.
XGS_AVR_NTSC_TILE_MODE5_V010.s|h	216x208, 36x26, **6x8** tiles resolution NTSC driver tile engine.
XGS_AVR_NTSC_TILE_MODE6_V010.s|h	144x208, 18x13, 8x8 tiles resolution NTSC driver tile engine with (5) 8x8 sprites.
VGA Bitmap and Tile Engines	
Bitmap Engines ►	
XGS_AVR_VGA_BMP_120_96_2_V010.s|h	120x96 2-bits per pixel (4 color) VGA graphics driver.
XGS_AVR_VGA_BMP_84_144_2_V010.s|h	84x144 2-bits per pixel (4 color) VGA graphics driver.
Tile Engines ►	
XGS_AVR_VGA_TILE_MODE1_V010.s|h	144x160, 18x20, 8x8 tiles resolution VGA driver tile engine.
XGS_AVR_VGA_TILE_MODE2_V010.s|h	144x120, 18x15, 8x8 tiles resolution VGA driver tile engine with **3** sprites.
XGS_AVR_VGA_TILE_MODE3_V010.s|h	144x120, 18x15, 8x8 tiles resolution VGA driver tile engine with **6** sprites.

Let's begin with a bit of a roadmap for the all the graphics drivers and libraries. Take a look at Figure 16.1 below.

Figure 16.1 – Graphics library & drivers architecture and relationship to applications.

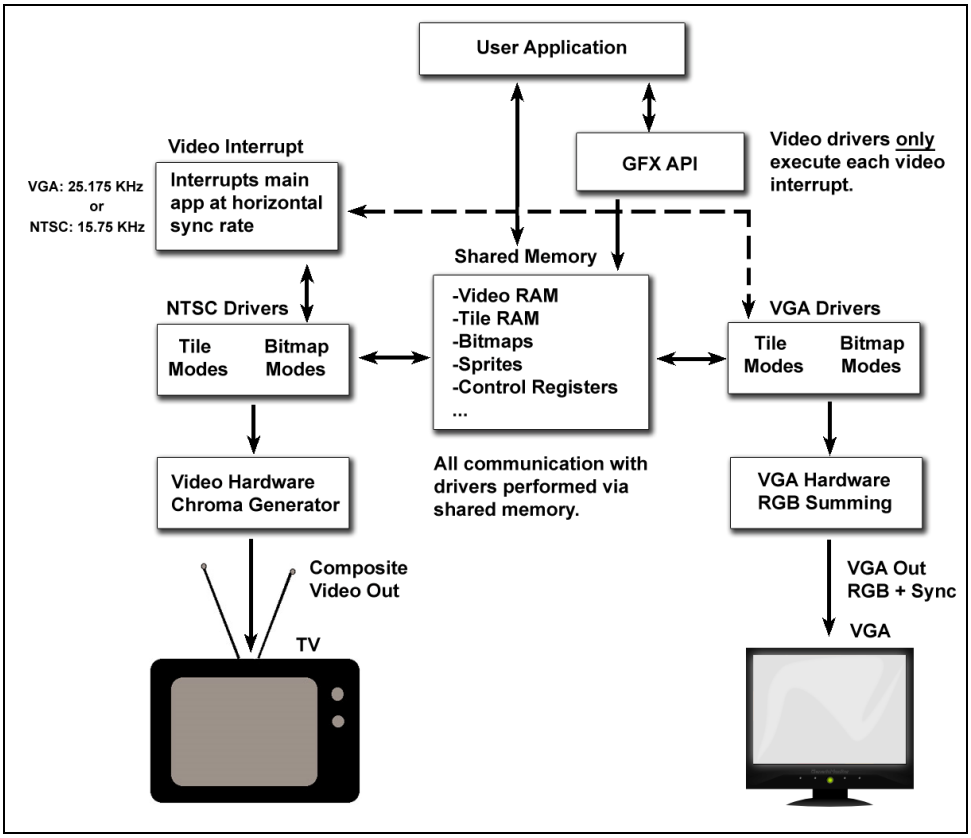
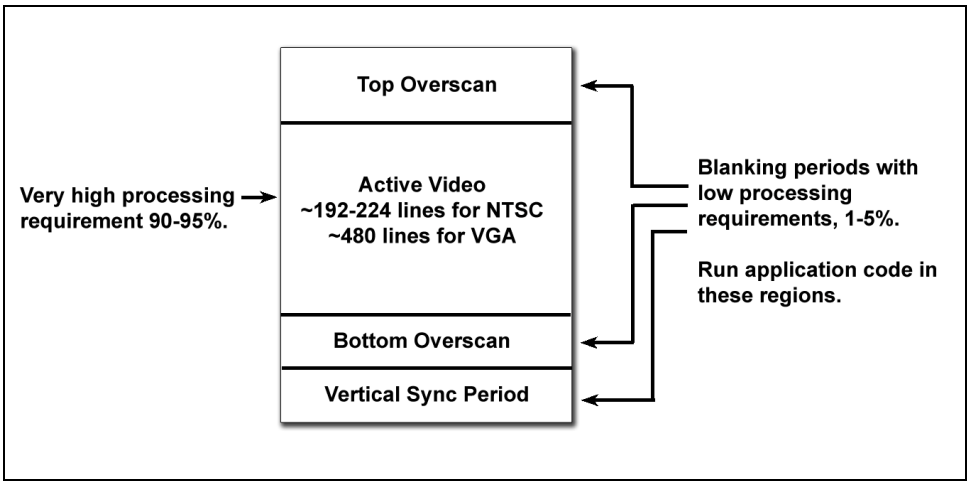


Figure 16.1 shows the relationship between the various graphics drivers and applications running on the XGS AVR. The idea here is that graphics are ***always*** generated by software via an **interrupt** that runs at the horizontal frequency scan rate of the video signal (**25.175 KHz** for VGA, **15.75 KHz** for NTSC approximately). The user level application initializes the graphics library which installs one of the graphics drivers (NTSC, VGA, bitmapped, or tile mapped). Then the user level application communicates with the graphics driver thru global variables, video RAM, and/or tile map memory. The idea here is that the graphics driver is running constantly generating video line by line. The processor during this is used up more or less 100% of the time. However, during the blanking periods and vertical retrace, the processor more or less is free to run application level code. This is shown in Figure 16.2.

Figure 16.2 – Application code runs during blanking periods for best performance and clean video.



The trick to making this work is the graphics drivers (all of them) must export out the current video line that the raster is on, this information coupled with the geometry of the graphics mode (exported in headers) is then used to infer if the raster scan is in the active video, blanking, retrace etc. With this information, the user level application can **"block"** or ***"wait"*** will the video is active and then once the video isn't rendering, the user application can run code.

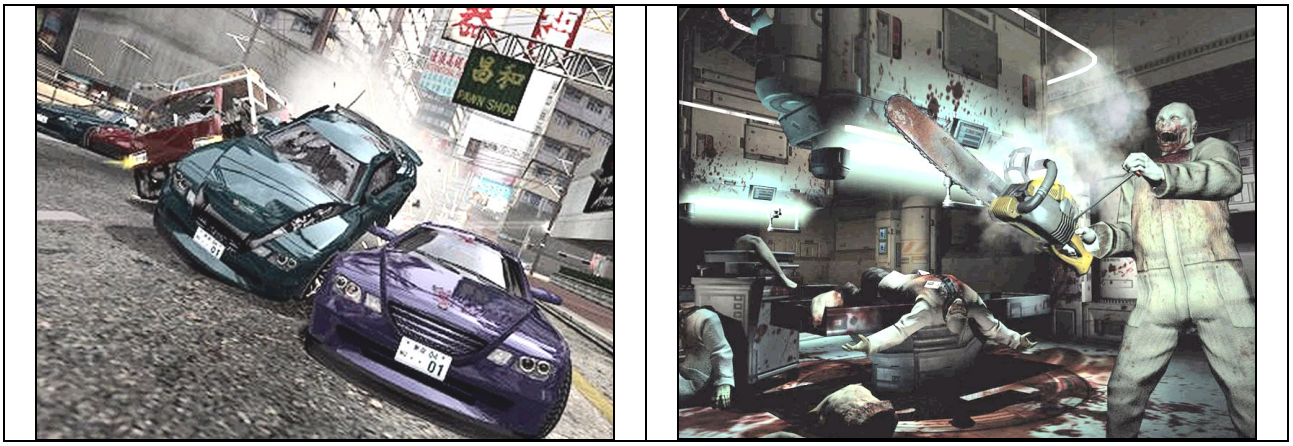
The reason you want to do this is if you run code while the video is being drawn, the interrupt driven video driver will exhibit "tearing" and distortions due to clock jitter on the interrupts. That is, when an interrupt is called there is no guarantee what the processor is doing and it must finish the current instruction. This might be an extra 1-2 clocks, thus, this small latency is just enough to make the image look fuzzy or unclean. Thus, user level applications will need to wait for the proper time to render on the screen and run code.

Now, looking back at Table 16.1 above there are a lot of graphics drivers available. The reason is that memory is so constrained on the AVR 644 (4K of RAM) that I decided to make as many permutations of drivers as possible, so that you can use them as a starting point for what your needs are specifically. That said, you will also notice a number of ***"classes"*** of drivers. There are both NTSC and VGA to begin with. The NTSC drivers of course generate NTSC signals for your TV set in composite format. While the VGA drivers generate VGA signals for your VGA compatible input devices. Both formats are important and have their uses. For example, if you are making games, you probably will like the NTSC modes more since they can run on any TV and you can carry your XGS AVR to your TV set to play. On the other hand, if you want to generate some very clean diagnostic displays or simulated controls then VGA might be a better fit.

Additionally, you see that there are both ***"bitmap"*** and ***"tile map"*** drivers for NTSC and VGA. If you're not a graphics expert then you might not have a very clear understanding of these terms, so let's take a moment to discuss each at a high level. then in the sections below we will discuss the technical points at length.

Bitmap graphics means that you have an image on a screen that you can individually address every pixel on the display. Figure 16.3 depicts a couple screen shots from bitmapped display games. As you can see, there is no regularity to the image, its got a lot of information and color, every single pixel is generated each frame and is individually addressable.

Figure 16.3 – A pair of 3D games that use bitmap display imagery techniques.



For example, in a standard VGA 640x480 16-bit color mode, the screen is 640 pixels wide, 480 pixels tall and each pixel is represented by 16-bits or 2-bytes. Thus, the total memory for such a display buffer is:

640 * 480 * 2 = 614, 400 bytes!

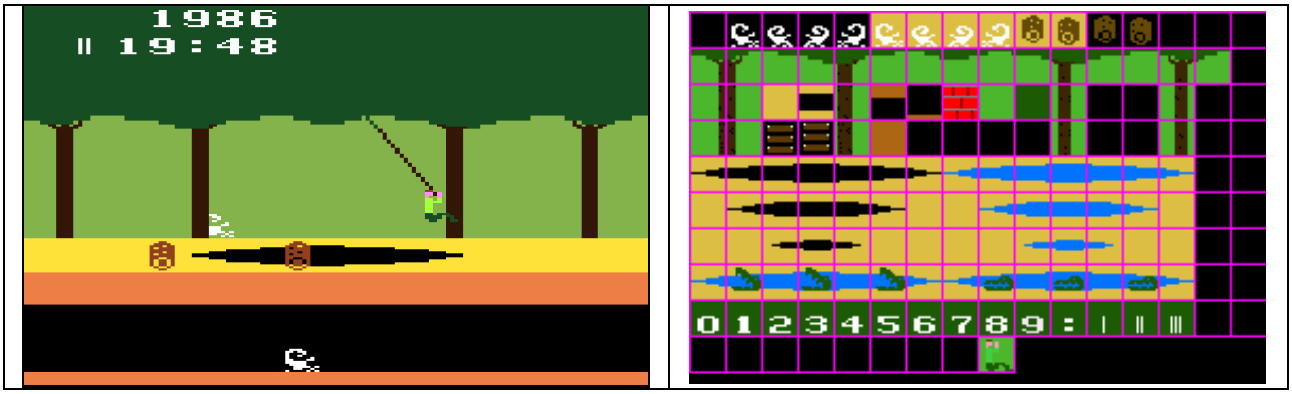
Which is quite a bit of memory even on a PC. Moreover, to facilitate animation you might need a ***"double buffered"*** display. In other words, while you are looking at one image, the computer generates the next on an off screen buffer, then during the vertical blank or retrace of the video system, the images are flipped or copied. This doubles the amount of memory you need! Therefore, bitmapped graphics on embedded systems like the XGS AVR will typically have much lower resolutions and typically use bits to encode colors rather than bytes. For example, the drivers that we develop represent pixels with 2-bits per pixel. This allows only 4 different pixel colors, but the memory requirements are much lower. For example, on the XGS one of the drivers is 120x96 with 2-bits per pixel or 4-pixels per byte, thus the memory requirements for this mode are:

120 * 96 / 4 = 2880 bytes!

Which is much better and fits in the 4K RAM of the AVR 644. Of course, you might be saying, ***"120 x 96?, my cell phone has a larger screen!"*** And you would be correct, but you would be surprised how well 120x96 can look on a TV with the use of color indirection or palettes to increase color depth (we will get to this later).

Moving on, the other primary class of computer graphics (at least in gaming and text based systems) is called ***"tile mapped graphics"***. Tile based graphics systems include many of the old 8-bit gaming computers like the NES, Sega Genesis, Atari 2600, and others. Also, many modern arcade games and PC games use tile based graphics. Tile based graphics beats the bitmapped memory constraints of small computers by using bitmaps, but in a tiled fashion, so the same tiles (bitmaps) are re-used over and over. Thus with only a few tiles, imagery can be generated that looks like it took vastly more memory to represent. Figure 16.4(a) below shows a tile mapped image while Figure 16.4(b) shows the tiles that make up the image.

Figure 16.4 – The Atari 2600 classic "Pitfall" along with some of the tiles templated.

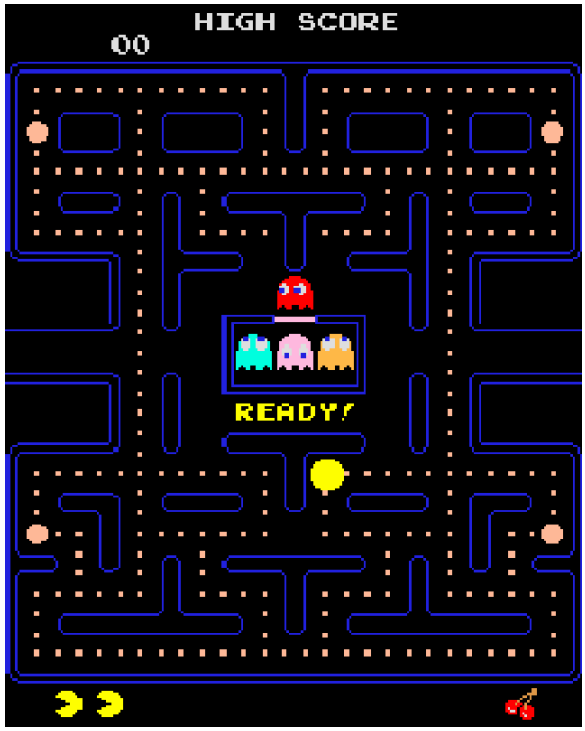


Referring to Figure 16.4, you can see how this system works. As an artist you might start by creating some sketches of the kinds of screen and levels you want then you might start drawing some mock ups of the imagery. Then at some point, you begin ***"ripping"*** the imagery into a set of common rectangular tiles usually 8x8 or 16x16 pixels in size and then template them into a ***"tile set"*** (Figure 16.4(b)) of bitmaps. Then from the set of tiled bitmaps you generate all your actual game screens. The memory savings comes from the indirection in this system. Instead of using the actual bitmap data, you use an index or pointer to the bitmap in the tile set, thus a tile map might consist of 40x24 tiles, each tile is represented by a single byte index (thus a total of 256 tiles can be represented) and each tile is 8x8 pixels, thus the final frame buffer image or bitmap image would be:

Width 40*8 = 320
Height 24*8 = 192

But, the amount of memory to represent the tile map would only be **40*24 = 960 bytes**! Of course, each tile bitmap still consumes precious memory, but you don't need that many of them. So depending on the game and how much data is redundant on each screen, you can get away with few bitmap tiles. This all sounds too good to be true? So, what's the catch? Well, the catch is that since the graphics modes are tiled. You can't place an image just anywhere you want, it has to be on the tile matrix aligned perfectly. Therefore, its very hard to move a character smoothly on the screen. Moreover, its very hard to move anything on TOP of the tile mapped image without replacing the tile at any particular location. Of course, there are tricks to do this, but they require ***"re-writing"*** the bitmap tiles themselves and using ***"tile animation techniques"***. A more common solution is the use of "sprites". A sprite is a bitmap image that is rendered ***on top*** of the tile mapped image; moreover, sprites move smoothly in most cases, support collision with each other and can be larger or smaller than the tile map bitmaps themselves. A good example of sprite use are say the ghosts in PacMan, the blue maze of PacMan is a tile mapped image, but the ghosts and PacMan himself are sprites, therefore, they can move freely around the image since they are rendered on top of the image without disturbing the tile map. Figure 16.5 shows a screen shot of PacMan for reference.

Figure 16.5 – A the famous game PacMan showing a tiled background with sprite based characters.



The tile mapped drivers developed support sprites as well, so you can implement games and demos with this type of functionality. In the next two sections we will investigate both bitmap and tile mapped mode in more detail as they relate to the XGS AVR drivers, so if you aren't a graphics guru yet, hopefully, the specific material below will clear it up. Since there are so many drivers, what we are going to do is start off with a generic overview and explanation of one of the primary drivers (bitmapped and tile mapped) then drill down and discuss each driver in the system along with any significantly different data structures. Of course, by design all the NTSC/VGA bitmapped drivers are very similar; once you understand one of them you understand all of them. Similarly with the tile drivers.

16.2 Bitmap Graphics Primer and Driver Overview

In this section, we are going to discuss the details about the bitmapped graphics drivers we have developed to get you started. All of the drivers are very similar, so we will focus on one driver more or less then make comparisons to the others where needed. The important take away from this section is to understand how bitmapped displays work, the relationship of the palettes, and the various data structures. Also, the NTSC and VGA drivers are nearly identical from a programmer's point of view. The differences are in how colors are encoded (which only matters during the initialization of the color palettes), as well as the resolutions are slightly different, and the sprite capabilities are slightly different. But, if you understand one driver, you more or less understand them all. To review, here are the current bitmap drivers in the XGS AVR library:

- XGS_AVR_NTSC_BMP_112_96_2_V010.s|h — 112x96 2-bits per pixel (4 color) NTSC graphics driver.
- XGS_AVR_NTSC_BMP_80_160_2_V010.s|h — 80x160 2-bits per pixel (4 color) NTSC graphics driver.
- XGS_AVR_VGA_BMP_120_96_2_V010.s|h — 120x96 2-bits per pixel (4 color) VGA graphics driver.
- XGS_AVR_VGA_BMP_84_144_2_V010.s|h — 84x144 2-bits per pixel (4 color) VGA graphics driver.

Let's go ahead and use the 1st driver **XGS_AVR_NTSC_BMP_112_96_2_V010** as the example for our discussion of both bitmap graphics and the driver itself, with that in mind, let's begin our discussion.

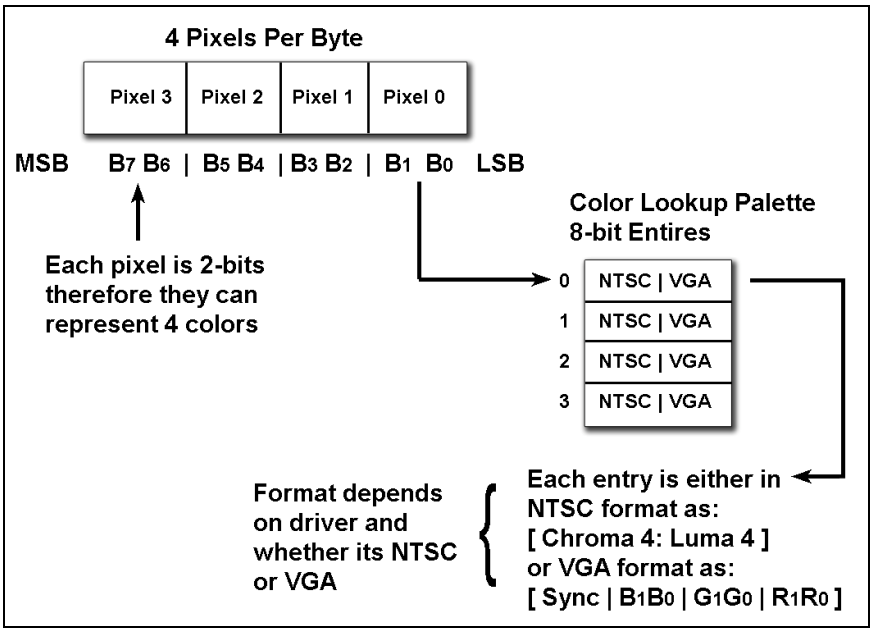
Each bitmap driver has a specific resolution on the screen. In this example, the resolution is **112x96** pixels. Additionally, each pixel is represented by 2-bits. Thus, each byte packs 4 pixels into it. Now, since the 2-bit pixels can encode 4 patterns as shown in Table 16.2.

Table 16.2 – Color encoding possibilities for 2-bits per pixel.

Color Index	Binary Bit Encoding
Color 0	00
Color 1	01
Color 2	10
Color 3	11

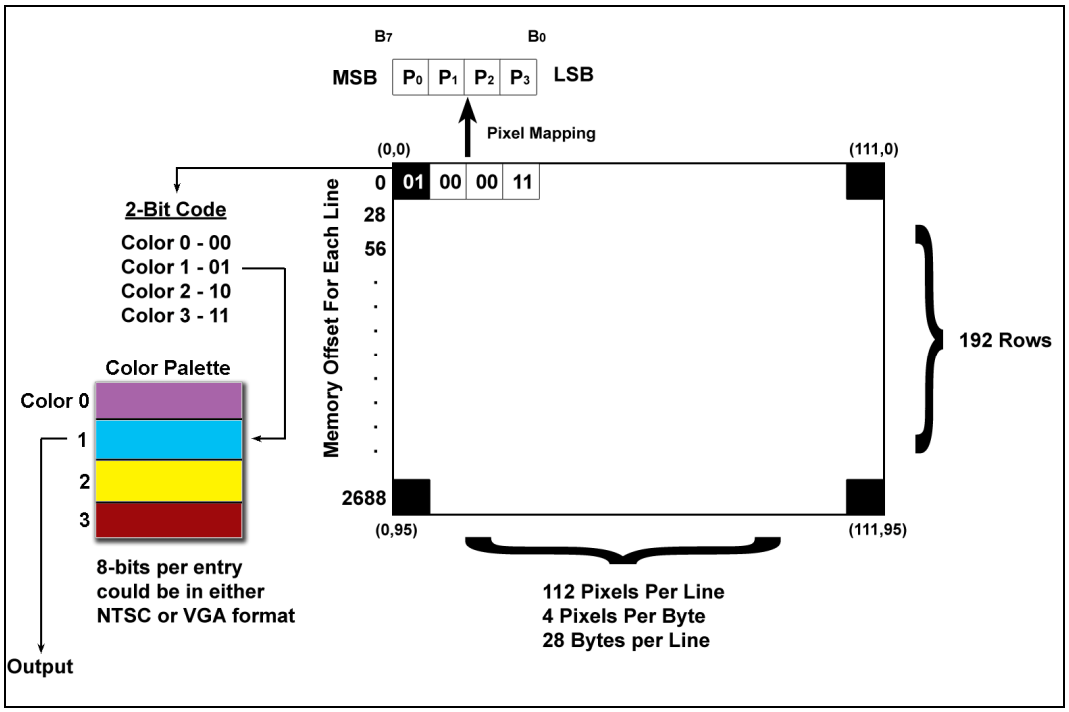
Moreover, as noted, each byte packs 4-pixels as shown in Figure 16.6.

Figure 16.6 – Each byte packs (4) 2-bit pixels LSB to MSB.



This color ***"indirection"*** gives us the ability to display ***any*** 4-colors we wish since the 2-bit color indices don't represent actual colors, but ***references*** to colors in another color lookup table (we will get to this shortly) this also makes the NTSC and VGA drivers easier to deal with since the color encoding in the bitmap doesn't change but only the last indirection before rendering out the signals. Finally, there is one last trick employed in the bitmap drivers and that's to use a tile map or zone map for different possible color palettes. In other words, we only get 4 colors, but we can use a different palette every 8 or 16 lines, or maybe every 8x8 or 16x16 pixels, so we can get more colors on the screen without using more bits per pixel. All these concepts are a lot to take in at once, let's clear it all up with Figure 16.7 which shows how everything ties together.

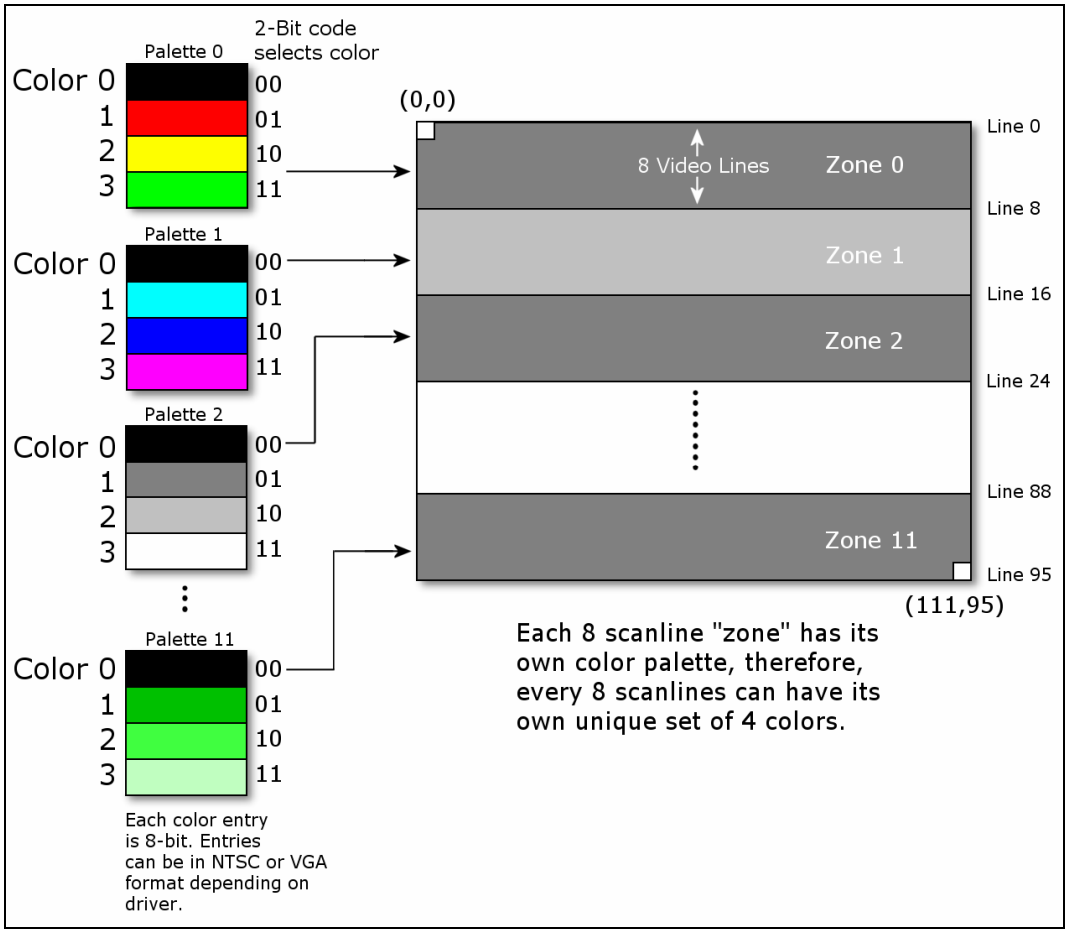
Figure 16.7 – Generalized architectural overview of the 112x96 NTSC bitmap mode.



There is a lot going on in the figure, so let's cover all of it. First, I want to bring your attention to the bitmap itself. It's 112x96 pixels, thus the coordinates are (0…111 x 0…95), with (0,0) being the upper left hand corner and (111,95) being the bottom right hand corner. This is a very important point to remember, bitmapped graphics unlike Cartesian Coordinates in high school are inverted, that is, X increases to the right, Y increases down. This is because the raster image is drawn top to bottom, thus its more natural for programmers to think this way. Additionally, memory addresses increase left to right, top to bottom, so address 0 would be the top left corner (0,0) and address (112x96/4 – 1) = 2879 would be the address of the bottom right corner (111, 95). This is a good transition into pixels addressing. Referring to Figure 16.7, another feature of the bitmapped mode is of course there are 4 pixels per byte, so not only do the addresses change left to right, but the pixel pairs do as well. But, we have to be a little careful here. There are two ways to do this; one way is to label the pixels left to right from msb to lsb and the other is to label them right to left lsb to msb (as the driver does). The later method is a little easier to deal with since the math works out better when you try to plot pixels and you need to locate a single pixel – not only do you have to locate the byte that the pixel is located in, you must locate the 2-bit pair that represent the pixel.

Continuing our analysis of the figure, another interesting feature shows how each pixel value (00, 01, 10, 11) all point to a ***"palette entry"*** that has the ***final*** and actual color that will be sent out to the screen. This is very important since there are two ways the drivers do this palette indirection. Method one supports a new palette or set of colors every 8-16 scanlines, some of the bitmap drivers use this method (including the NTSC 112x96 driver). On the other hand this is limiting since the same 4 colors are available for 8-16 lines (depending driver). The other palette method uses a tile map of sorts that gives every 8x8 or 16x16 set of pixels its own palette (this is how the NES works for example), this is much more flexible. The 80x160 NTSC bitmap driver supports this mode. The two different palette modes are shown in the left bottom and right bottom portions of Figure 16.7 above, but let's drill down a bit on this subject. First, let's talk about the ***"zone"*** palette method supports a new palette every 8 scanlines, this is shown in Figure 16.8, the 112x96 NTSC bitmap driver supports this.

Figure 16.8 – Palette zones every 8 scanlines.



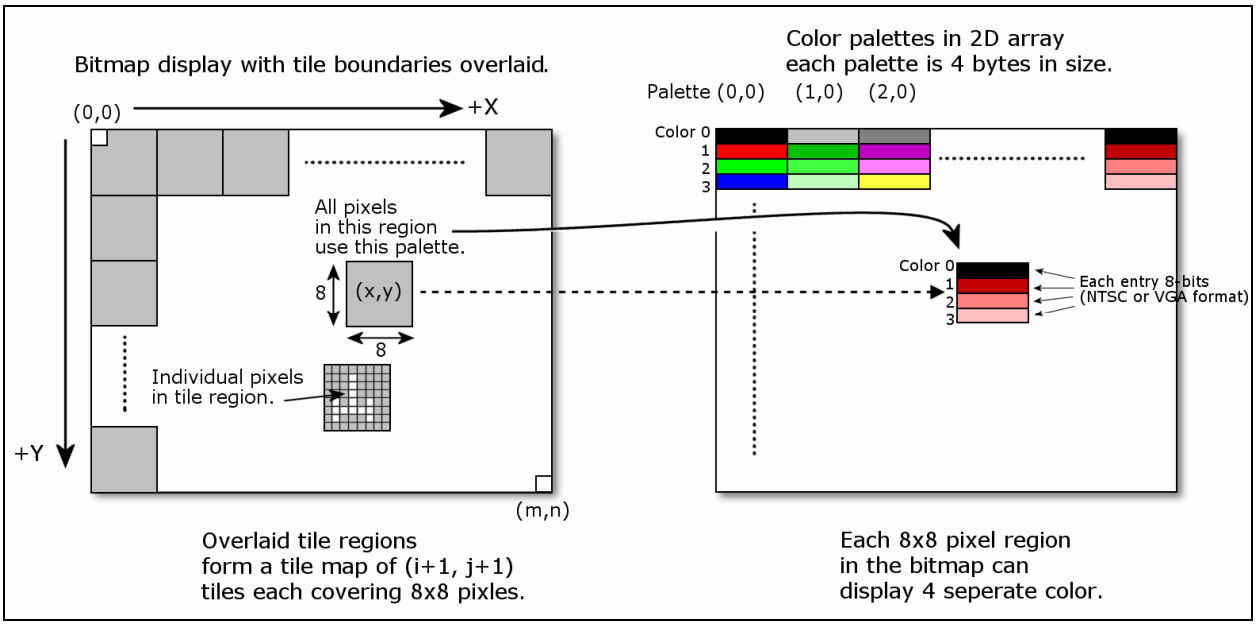
Referring to Figure 16.8, the idea is that you only have 2-bits per pixel, but every 8 scanlines you get another set of 4 colors, so you can change things up. Also, in the figure you see how the actual palette entries are encode for both NTSC and VGA. In the case of NTSC each palette entry is 8-bit, but encodes 4-bit **LUMA** and 4-bit **CHROMA**, in the case of VGA each byte encodes a 6-bit **RGB** value. The nice thing about this is that 8-bit entry is that it's abstract ***"data"*** as far as we are concerned as long as a particular entry has "red" in the context of the driver we are using. For example if color index 2 refers to "red", we can write a NTSC program or VGA program that both have nearly the same high level commands, and the same bitmap data, the only thing that differs is the palette data itself. An example of the default palette for the 112x96 zoned NTSC palette is listed below right out of the ASM driver file:

```
// these values arbitrary and just something to start with,
// the caller will want to initialize the palette himself

palette_table:

.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 0
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 1
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 2
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 3
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 4
.byte NTSC_BLACK, NTSC_GRAY1, NTSC_GRAY2, NTSC_GRAY3    // palette 5
.byte NTSC_BLACK, NTSC_GRAY2, NTSC_GRAY3, NTSC_GRAY4    // palette 6
.byte NTSC_BLACK, NTSC_GRAY3, NTSC_GRAY4, NTSC_GRAY5    // palette 7
.byte NTSC_BLACK, NTSC_GRAY4, NTSC_GRAY5, NTSC_GRAY6    // palette 8
.byte NTSC_BLACK, NTSC_GRAY5, NTSC_GRAY6, NTSC_GRAY7    // palette 9
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 10
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 11
```

If we were using the VGA driver then these values would change to their VGA RGB cousins, but that's it.

In any event, zoned palettes are a good start, but if you want even more flexibility then you might decide that you want to ***"tile"*** the bitmap with palettes, so that every m x n pixels you get a region of 4 new colors. This is exactly what the 80x160 NTSC bitmap driver does as well, it overlays a tile set of palettes that each cover a 16x16 pixel region. This is shown in Figure 16.9.

Figure 16.9 – Tiled palettes are much more flexible.



Referring to Figure 16.9, this is a much better system since you can change the colors a lot more frequently. Of course, the downside is that the palette memory increases. In this case, the resolution is 80x160, and we are using a new palette every 16x16 pixels, and each palette is 4 bytes, thus we need the following amount of memory:

160/16 * 80/16 * 4 = 200 bytes.

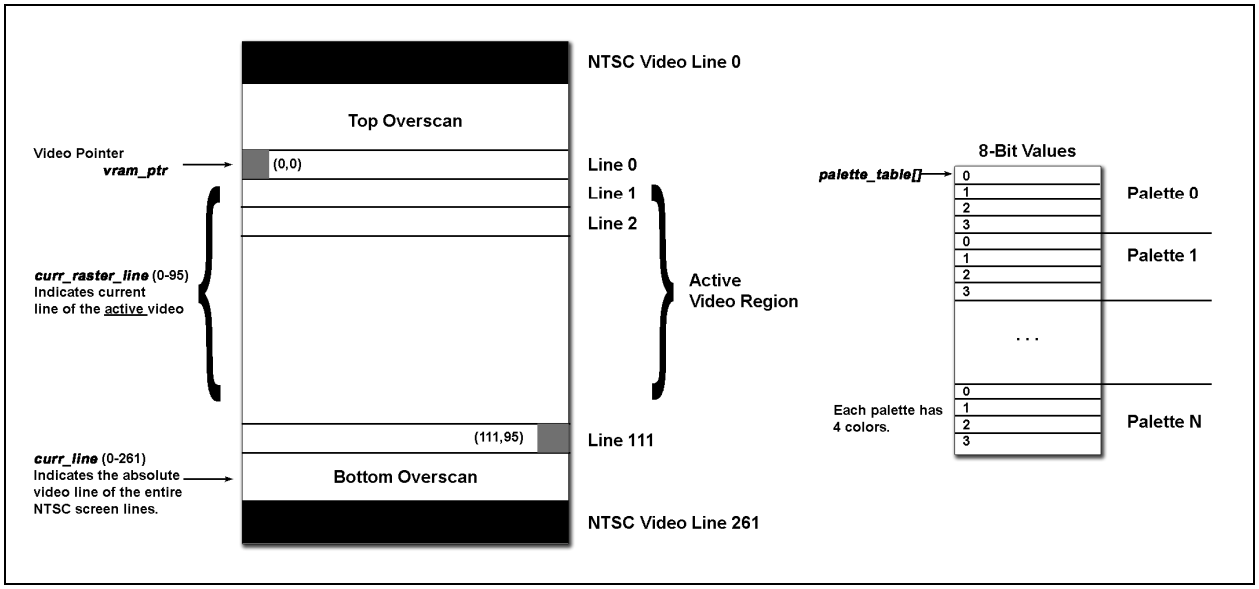
This isn't too bad, but it's definitely more memory than the zoned palettes which need only 12*4 = 48 bytes with the 112x96 driver.

Let's review now, before moving forward. There are both NTSC and VGA bitmap drivers. They are both roughly the same architecturally; they both use 2-bits per pixel which are color references, not the actual colors. The colors are always in a color table or palette and there are two different palette designs; zoned and tiled. Additionally, the bitmap buffer is a flat continuous region of memory and the coordinate (0,0) represents the top left pixel and also address 0. However, since pixels are packed 4 to a byte, you have to not only compute the proper address of a pixel, but the proper 2-bit pixel pair. Additionally, addresses increase left to right, top to bottom. Finally, each palette entry is 4 bytes and is either a chroma, luma pair representing an NTSC color or a RGB value, in both cases encoded as 8-bits. The drivers themselves deal with generating the video output either NTSC or RGB/VGA, so that you as a programmer are insulated from the details. You only have to worry about dealing with NTSC colors or RGB colors when you set up your palettes. With that all in mind, let's look at the exact data structures involved, their common names and how memory is accessed exactly in the next section.

16.2.1 Accessing the Bitmap and Manipulating Pixel Data

Let's continue our discussions by deconstructing the 120x96 NTSC bitmap driver as a model. The other bitmap drivers are all very similar, so once you understand one driver you understand them all. To begin with, take a look at Figure 16.10 below it illustrates the 112x96 graphics mode along with some data structure labels.

Figure 16.10 – The 112x96x2 (4) color NTSC bitmap graphics mode.



As you can see there are a number of data variables and pointers that are labeled in the figure. These are all part of the ASM driver and found in the file **XGS_AVR_NTSC_BMP_112_96_2_V010.s**. Here's an excerpt from the ASM file that lists the variables in question:

```
.section .data

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// NOTE: Each palette represents 8 scanlines on the screen, there are a total of 12 palettes,
// each palette holds 4 colors therefore 12*4 = 48 bytes total storage needed. Also, there is no requirement
// that the palette data must be on a page boundary, since there is time for full 16-bit addressing in the
// palette code.
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// these values arbitrary and just something to start with,
// the caller will want to initialize the palette himself

palette_table:

.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 0
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 1
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 2
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 3
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 4
.byte NTSC_BLACK, NTSC_GRAY1, NTSC_GRAY2, NTSC_GRAY3    // palette 5
.byte NTSC_BLACK, NTSC_GRAY2, NTSC_GRAY3, NTSC_GRAY4    // palette 6
.byte NTSC_BLACK, NTSC_GRAY3, NTSC_GRAY4, NTSC_GRAY5    // palette 7
.byte NTSC_BLACK, NTSC_GRAY4, NTSC_GRAY5, NTSC_GRAY6    // palette 8
.byte NTSC_BLACK, NTSC_GRAY5, NTSC_GRAY6, NTSC_GRAY7    // palette 9
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 10
.byte NTSC_BLACK, NTSC_RED, NTSC_GREEN, NTSC_BLUE       // palette 11

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// current video line
curr_line:              .byte 0,0

// current raster line 0..95
curr_raster_line:       .byte 0

video_state:            .byte 0     // state of video kernal 0=init, 1=run normally

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DATA STRUCTURES IN FLASH/ROM - NOTE: MUST ON WORD BOUNDARIES
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

.section .text

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// EXTERNALS
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////

.extern vram_ptr // 16-bit pointer passed in from caller that points to video RAM

////////////////////////////////////////////////////////////////////////////////////////////////////////////
// GLOBALS EXPORTS
////////////////////////////////////////////////////////////////////////////////////////////////////////////

// export these to caller, so he can interrogate and change
.global curr_line        // read only, current physical line
.global curr_raster_line // read only, current logical raster line
.global palette_table    // read/write, 16-bit base address to palette used for driver
```

As I said, this is only an excerpt and if you look in the ASM file itself there is more here. But, let's focus on relating the declarations to the labelings in Figure 16.10. First, there is the bitmap data buffer itself referred to as ***vram_ptr***, this is declared as an ***"External"*** in the ASM file, thus must declare it in one of your C files as a pointer in SRAM. The ASM file will link to this name and assume bitmap data for rendering is in it.

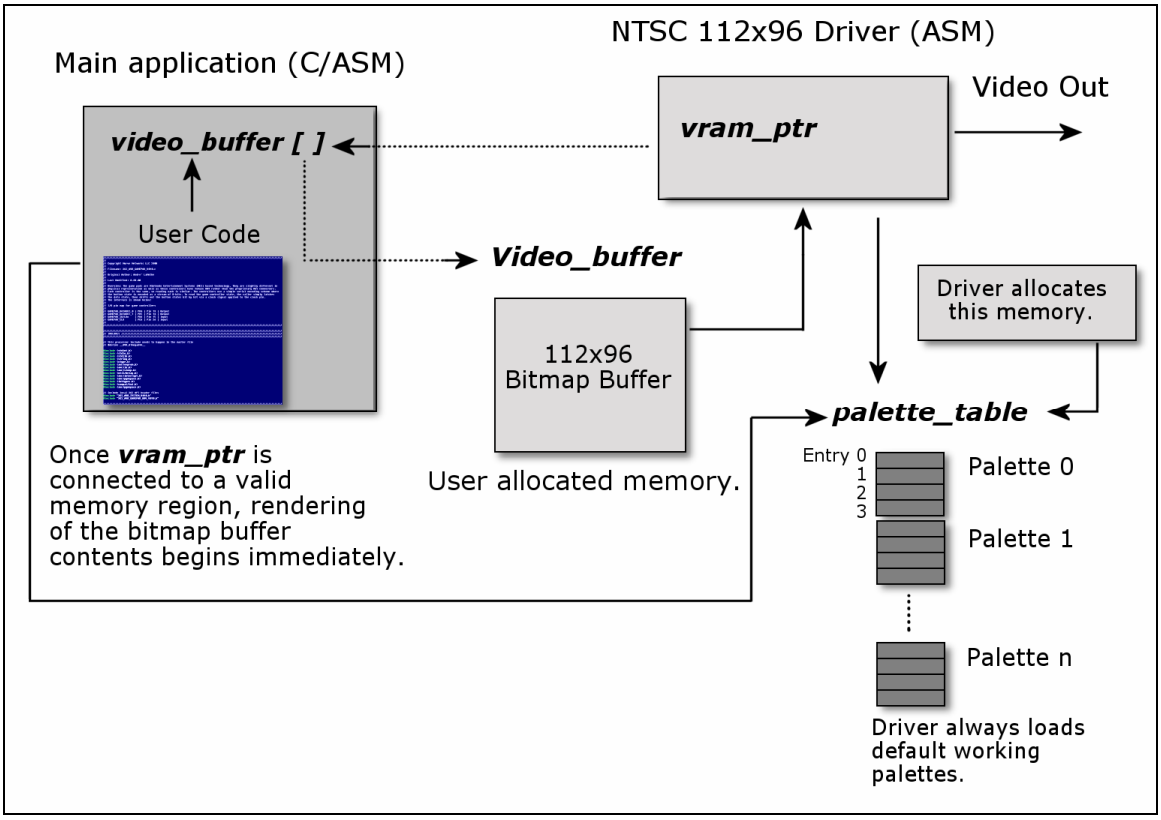
Next, in the ***"Global Exports"*** section there are three very important variables;

curr_line This is a 16-bit integer that tracks the "current line" of the video display. This is the actual NTSC or VGA line from 0..261 or 0..524.

curr_raster_line This is a 16-bit integer that tracks the "virtual" current line of the video mode. For example, in the 112x96 mode, this variable would track from 0..95 during the active display.

palette_table This is the starting address of the locally defined color palette for the mode. We could have defined it in the C application, but due to addressing tricks, we need it declared here. Read the paragraph above the palette declaration for comments on addressing and memory issues that are important with this data structure. In short, the palette ***must*** exist within a ***"page boundary"*** of 256 bytes for the addressing math to work.

Figure 16.11 – Interfacing with the 112x96 NTSC driver.



Interfacing with the driver is very simple as shown in Figure 16.11. The main C application need only declare ***vram_ptr*** as a ***unsigned char ****, then the other variables are available from the linked ASM driver by name without any mangling. Of course, enough memory has to be allocated to fit the bitmap buffer. This is calculated by the width and height and divided by 4 since there are 4 pixels per byte. Therefore, a sample declaration of ***vram_ptr*** might look like:

```
// memory storage for video buffer, notice / b 4 since 4 pixels per byte
volatile unsigned char video_buffer[112 * 96 / 4];

// externally linked symbol that ASM driver expects, point to video buffer
volatile unsigned char *vram_ptr = video_buffer;
```

That's all there is to it. Once this data structure is allocated then ***curr_line*** and ***curr_raster_line*** can be interrogated at any time by your C code to help you keep track of where the raster is. Additionally, palette_table can be read from or written to and you can update any of the palette entries (12 sets of 4) that you wish. This would change the colors you see every 8 lines (in this case) to a new set of colors. Of course each color is 8-bits and in the following NTSC format:

[LUMA3..0 : CHROMA3..0]

If this were one of the VGA drivers, then each color palette entry would be in the following RGB format (notice the data is actually in BGR format left to right):

[0 0 |B1 B0| G1 G0| R1 R0] - 6-bit color, upper 2-bits 0.

So at any time you can modify the palette entries and change colors on the screen on the fly. To recap, the steps you need to use the bitmap drivers:

- You simply link with one of the NTSC or VGA drivers, in our example the 112x96 NTSC driver.
- In your C application you need to declare a single unsigned char pointer variable ***vram_ptr***. This will be the interface between the C application and the ASM driver.
- You write pixel data to the screen buffer, update palette entries, and interrogate the line tracking variables.

Of course, we are leaving the details out of initializing and installing the video driver and interrupt, but we will get to that later. Right now, let's continue to develop how the driver(s) work.

16.2.2 Plotting a Pixel Bitmapped Modes

The most important thing you understand is how to plot a single pixel in the bitmap modes. Once you have this mastered, then you can draw lines, polygons, and create 3D scenes even, but it all starts with the pixel. Let's begin with a hypothetical example of plotting pixels in a byte per pixel video mode (which we don't have). If every pixel is represented by a single byte, and video memory is continuous from left to right, top to bottom then computing the address of any (x,y) pixel is very easy. First, let's compute the starting address of the y row we are on (assuming a 112x96 pixel mode):

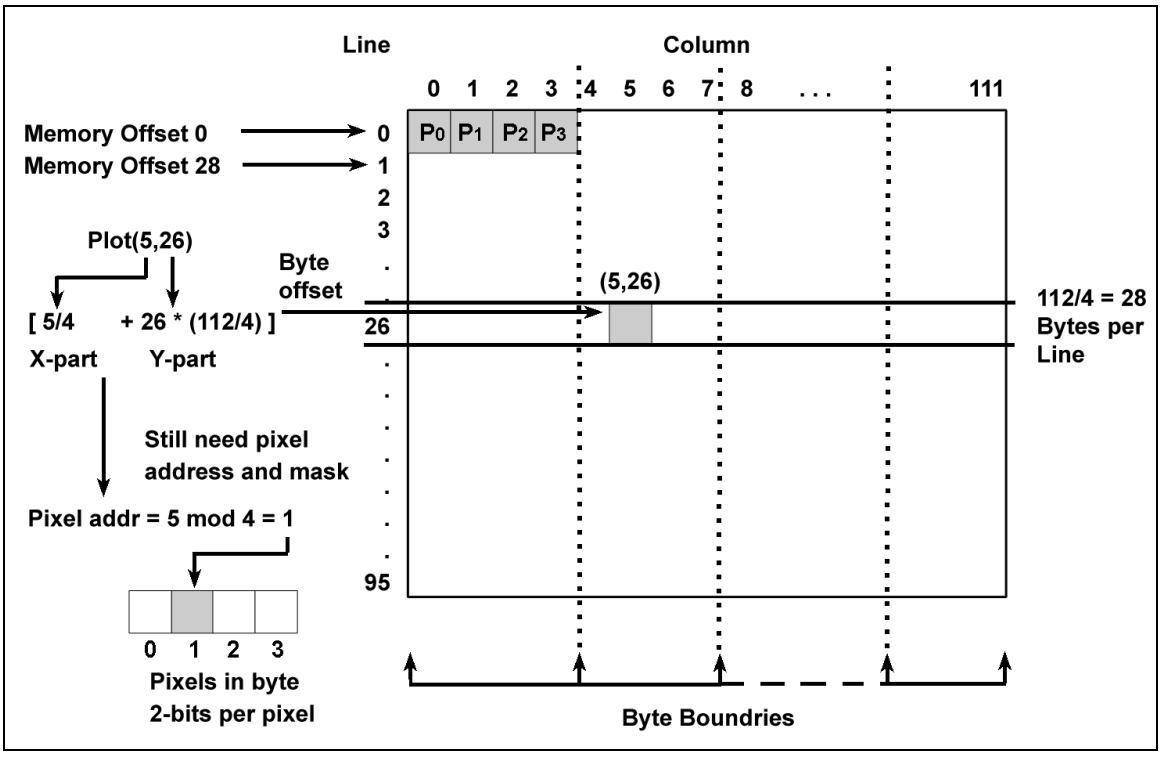
```
pixel_addr = y*112;
```

Wow, that was easy! Now, let's take x into consideration:

```
pixel_addr = x + y*112;
```

Thus, the pixel address is nothing more than the x added to the y multiplied by the ***"memory pitch"*** that is, how many bytes per line. Figure 16.12 shows this computation for our hypothetical graphics mode with 1 byte per pixel.

Figure 16.12 – Computing pixel addresses for 1-byte per pixel graphics mode at 112x96.



But, we are using a 4 pixel per byte graphics mode… This reduces memory by a factor of 4, but makes computing pixel addresses a little more complex. But, let's start with what we know. First, the new **"memory pitch"** is **112/4**, so to compute the y contribution to pixel address is as follows:

pixel_addr = y*112/4;

Not, too bad, but what about the x contribution? Well, a little thought, and you will conclude it's the same calculation with a division by 4:

pixel_addr = x/4 + y*112/4;

And those readers with an eye toward optimization will realize that we can simplify this to:

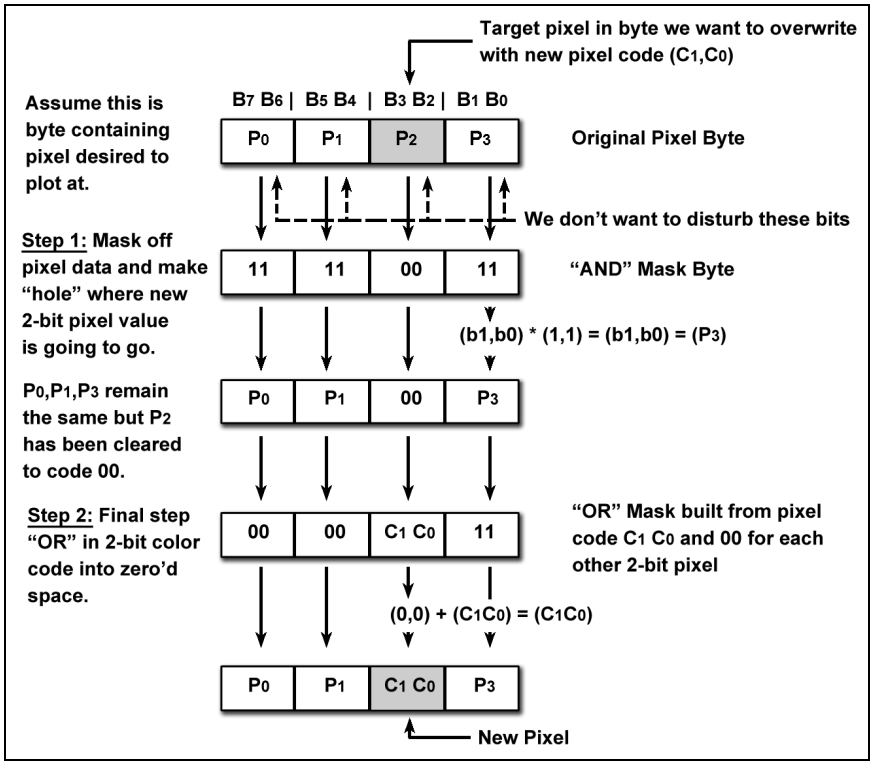
pixel_addr = (x + y*112) / 4;

And we can remove the division with a shift to the right:

pixel_addr = (x + y*112) >> 2;

Referring to Figure 16.13 below, we are close, but still we have one big problem and that's locating the pixel within the byte, and then masking the pixels already in the byte. Take a look at the figure to see what I mean.

Figure 16.13 – Plotting a single pixel with packed 2-bit per pixel data.



Looking at the graphical representation of the algorithm in Figure 16.13, there are some bit manipulations that must be performed. First, we must locate the pixel 2-bit pair in the byte. This is done with a simple mod operation:

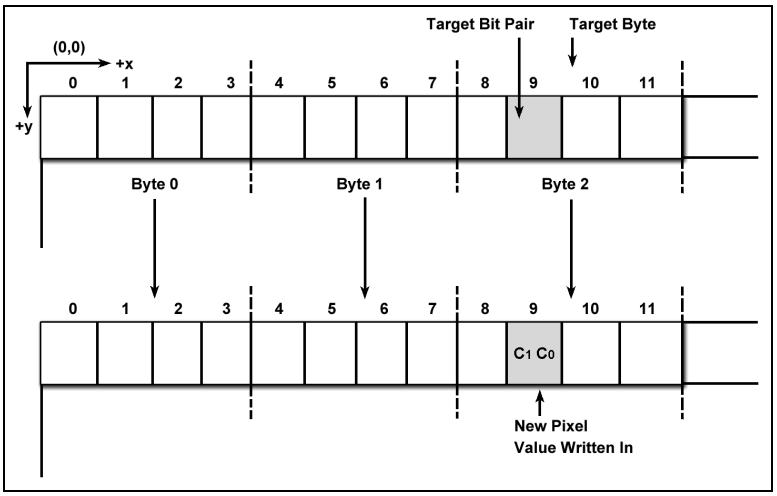
pixel_shift = 2*(x mod 4);

pixel_shift would then represent the number of times to shift left the 2-bit color code "c1c0". As an example, say that we want to plot a pixel at location (0,y), then the calculation would result in:

pixel_shift = 2*(0 mod 4) = 0

This makes sense since x=0 is represented by the first pair of bits in the first byte (y is irrelevant). Now, let's try another example, (9,y). In this case, we know that there are 4 pixels per byte, so the pixel must lie in the 3rd byte, 2nd position as shown in Figure 16.14(a).

Figure 16.14 – Plotting (9,y).



Let's try our calculation and see if the bit shift works:

pixel_shift = 2*(9 mod 4) = 2

In other words, we should shift the initial color value "c1c0" exactly one 2 positions to the left resulting in the image shown in Figure 16.14(b) (remember we render the pixels LSB to MSB in pairs of 2). Hopefully, you see that our algorithm works. The remaining problems are with ***"masking"*** issues. Referring back to Figure 16.13, not only do we have to create the proper byte data, but we have to mask a ***"hole"*** for it to be placed into and OR it into the current data. This is important since we do not want to disturb the other pixels already there. This whole operation takes quite a few bit manipulation operations, but is straightforward more or less. Below for reference is the listing of one of the actual bitmap plotting functions from the library that perform this operation completely.

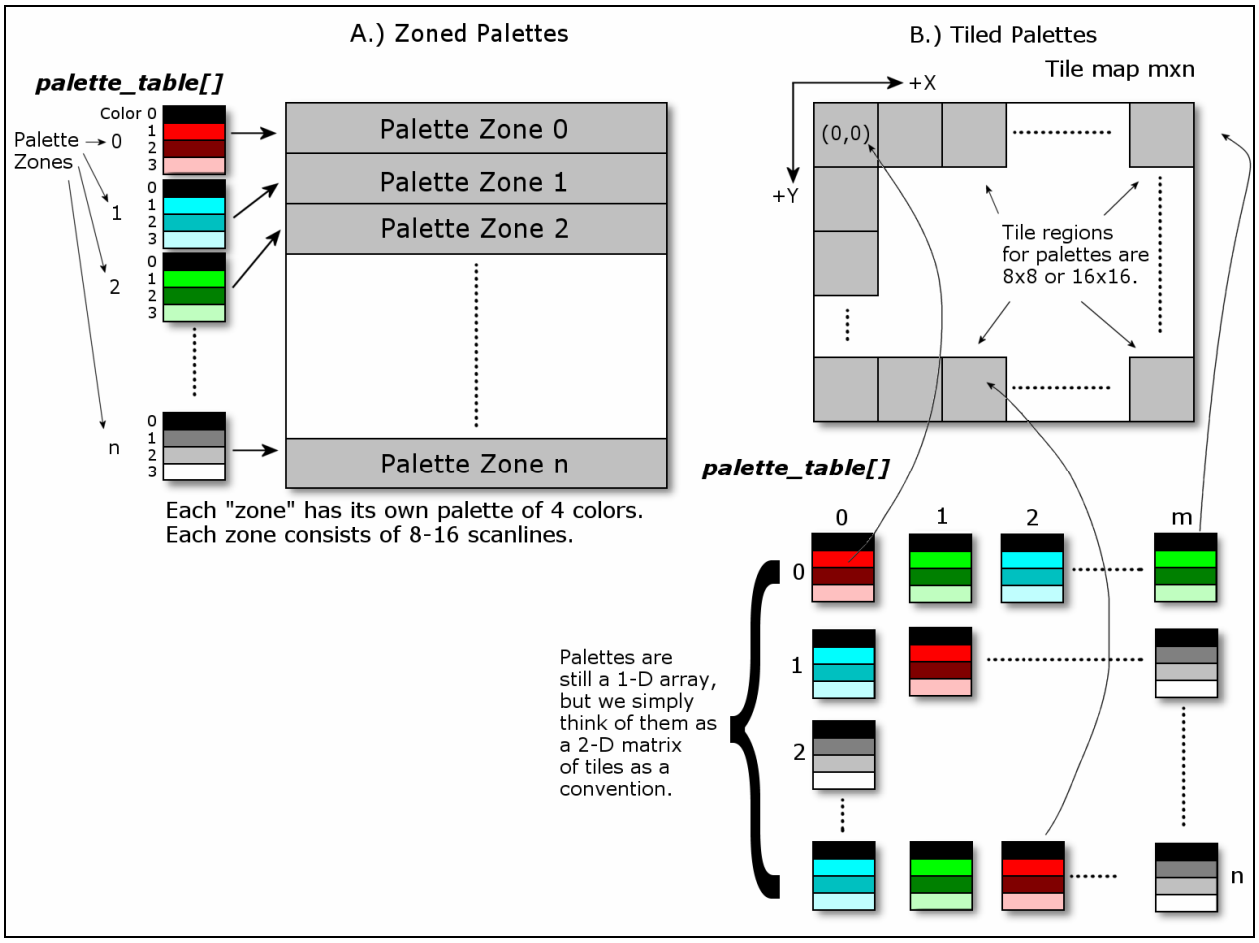
```
inline void GFX_Plot(int x, int y, int c, UCHAR *vbuffer)
{
// plots a pixel on the screen at (x,y) in color c
// there are 2 bits per pixel, thus to locate the pixel, we need to divide the x by 4 and add //
// that to the y * SCREEN_WIDTH/4, then that byte contains the pixel,
// next we need to shift the color over 0..3 pixels over
// where each pixel is 2-bit, and AND/OR mask it into the video byte
// Note: later convert to pure ASM, also try optimizing common terms (compiler should do this?)
// shifts are slower than multiples, so on AVR the mutliplications should be faster?

vbuffer [ (x >> 2) + (y * (g_screen_width >> 2) ) ] =
        (vbuffer [ (x >> 2) + (y * (g_screen_width >> 2)) ] & ~(0x03 << 2*(x & 0x03)) ) |
        (c << 2*(x & 0x03));

} // end GFX_Plot
```

As you can see some nasty bit manipulations along with some optimized code, but it does what it needs to do.

16.2.3 Manipulating the Color Palettes

The color palette support in the bitmap modes is the last element we need to discuss. There are two different kinds of bitmap drivers NTSC and VGA, both drivers have palette support either zoned palettes that support 4 new colors per 8-16 video lines, or the tiled palettes which support 4 new colors every 16x16 pixel region. Depending on which driver you are using and what type of palette it has (zoned or tiled) you will always refer to the palette data as ***palette_table***. All bitmap drivers declare and export this symbol out, so you can use it always. Moreover, its always a byte array (unsigned char) where its organized as a set of 4 byte palette entries that either represent rows on the screen, or tiles in a 2D map of 16x16 regions of color change. For example, let's say that you are using a bitmap mode that supports zoned rows of palettes, each palette always has 4 color entries mapping to the color pixel codes of 00, 01, 10, 11 (0,1,2,3) this is shown in Figure 16.15(a).

Figure 16.15 – The organization of zoned and tiled palettes.



Let's say that you wanted to update palette index 3 representing the 4th entry and write all 4 colors:

```
// each palette entry is a single byte, each palette is 4 bytes
palette_table[3*4 + 0] = color1;
palette_table[3*4 + 1] = color2;
palette_table[3*4 + 2] = color3;
palette_table[3*4 + 3] = color4;
```

Where ***color1***,2,3,4 might be in NTSC format or VGA format.

TIP	The palette is being accessed as a continuous array. But, there is no reason why you can't create a new C ***TYPEDEF*** with 4 bytes, then a pointer to this type and assign it to the ***palette_table***. Then you could potentially access palette entries in the format: palette_data[index].entry0

As the next example, refer to Figure 16.15(b), this shows the tiled variation of palettes, but the thing to remember is this is simply an interpretation of the data. The palettes are still 4 byte entries in a single array, thus, we simply need to access them as if they were a 2D matrix representing the tiles (very similar to the bitmap data itself; 2D on the screen, but 1D in memory). For example, in the 80x160 NTSC bitmap driver, it uses 16x16 pixel palette tile map regions, thus there are 10 rows of 5 palettes, or a 5x10 tile palette. Therefore, to access any (x,y) palette and manipulate the 4 entries, the following code can be used:

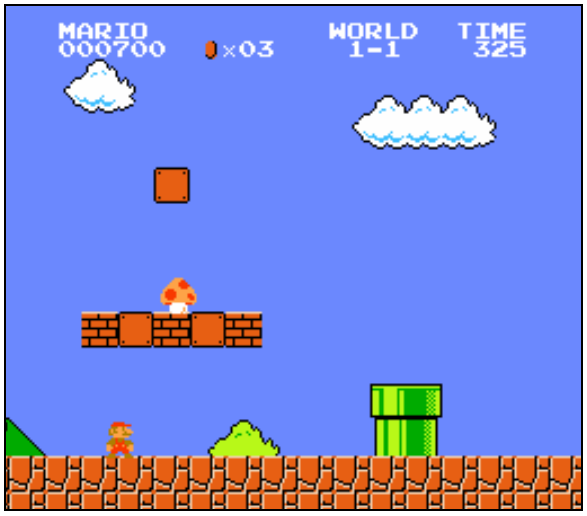
```
// each palette entry is a single byte, each palette is 4 bytes, the tiled palettes are 5x10 in
the example
// notice the address calculation, there are 5 rows and each palette is 4 bytes
palette_table[(x + y*5)*4 + 0] = color1;
palette_table[(x + y*5)*4 + 1] = color2;
palette_table[(x + y*5)*4 + 2] = color3;
palette_table[(x + y*5)*4 + 3] = color4;
```

In conclusion, even with 2-bits per pixel bitmap graphics and low resolutions, you can create striking imagery and numerous graphics and game applications. The addition of color indirection via palettes (zones and tiled) adds a whole other level to the potential richness in color to the images that can be generated. In the right hands, a good programmer can code in such a way that users will think its full 8-bit color or more!

16.3 Tile Mapped Graphics Primer and Driver Overview

Tile mapped graphics are the second main class of 2D computer graphics that you might commonly see used for games that aren't 3D. The idea of tile mapped graphics and outlined earlier is that you don't control every single pixel on the screen directly, but you control ***"bitmap tiles"*** that are usually 8x8, 16x16, or 32x32. The image on the screen is a 2D array of these tiles much like a text display is. For example, when you are typing in a text mode on a DOS or Linux shell the screen might be 40x24 or 80x50 characters, each character might be 8x8 or 10x12, etc. But, the point is that each character is a "tile" they just happen to look like characters from the English alphabet. But, they could just as well be monsters, textures, weapons, etc. This is the idea of tile mapped graphics to represent interesting images by re-using the same bitmaps over and over. Figure 16.16 shows one of the most popular tile mapped games in history ***"Super Mario Brothers"***.

Figure 16.16 – A screen shot of Super Mario Brothers for the NES.



Referring to the screen shot, you can immediately see the **"tiles"** in the image that are repeated. Examples are the blue sky, the ground, the brick, the clouds are a couple tiles, and so forth. This is the idea of tile mapping. If you look back to Figure 16.4 we saw the Atari ***"Pitfall"*** example with a screen shot and the tiles that were the source of the image as another example.

Therefore, you can see tile map engines are very powerful and commonly used in memory constrained games (8-bit and retro) and simply where bitmapped graphics make no sense. For the XGS AVR, we have developed a number of tile mapped drivers for both NTSC and VGA, some with sprites, some without. Among all the engines you should find something that meets your needs or something that you can use for a starting point to develop a new engine with the specifications that you desire. There are a lot of tile engines in the XGS AVR library, so let's briefly review the list:

NTSC Tile Engines

- XGS_AVR_NTSC_TILE_MODE1_V010.s|h 160x192, 20x24, 8x8 tiles resolution NTSC driver tile engine.
- XGS_AVR_NTSC_TILE_MODE2_V010.s|h 180x192, 22x24, 8x8 tiles resolution NTSC driver tile engine.
- XGS_AVR_NTSC_TILE_MODE3_V010.s|h 208x208, 26x26, 8x8 tiles resolution NTSC driver tile engine.
- XGS_AVR_NTSC_TILE_MODE4_V010.s|h 240x208, 30x26, 8x8 tiles resolution NTSC driver tile engine.
- XGS_AVR_NTSC_TILE_MODE5_V010.s|h 216x208, 36x26, 6x8 tiles resolution NTSC driver tile engine.
- XGS_AVR_NTSC_TILE_MODE6_V010.s|h 144x208, 18x13, 8x8 tiles resolution NTSC driver tile engine with (5) 8x8 sprites.

VGA Tile Engines

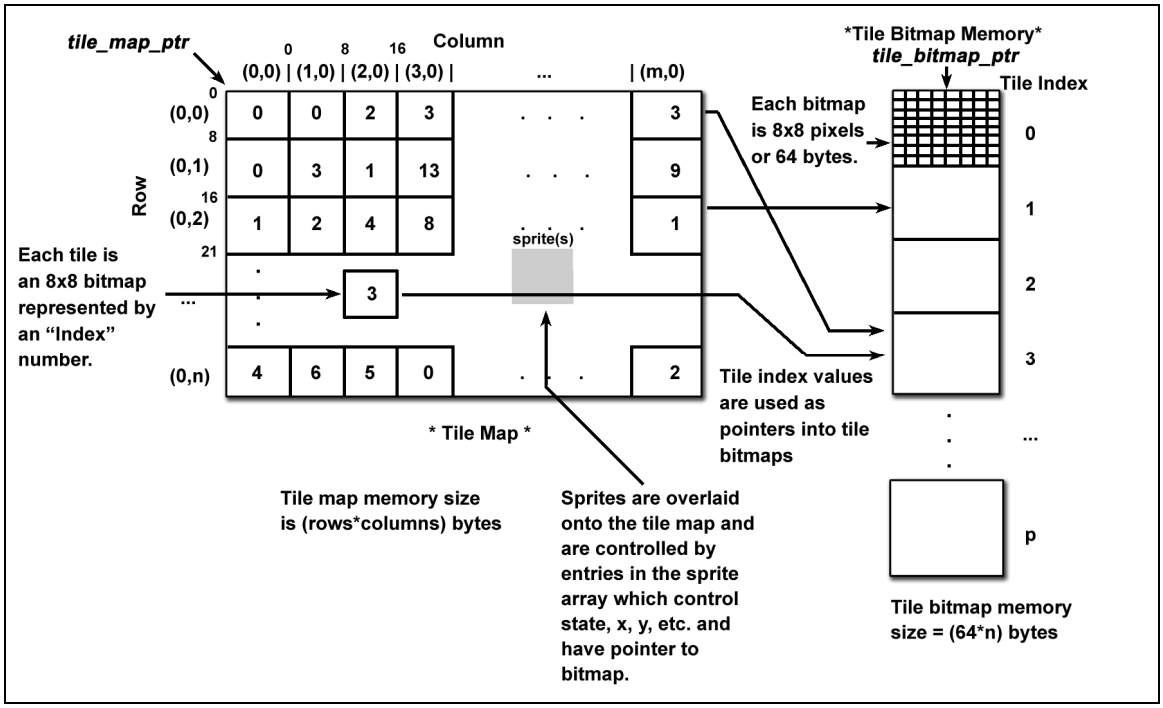
- XGS_AVR_VGA_TILE_MODE1_V010.s|h 144x160, 18x20, 8x8 tiles resolution VGA driver tile engine.
- XGS_AVR_VGA_TILE_MODE2_V010.s|h 144x120, 18x15, 8x8 tiles resolution VGA driver tile engine with 3 sprites; ; 2 full transparent, 1 opaque with scaling. Supports sprite clipping left, right, bottom, top NOT supported, must be done by caller.
- XGS_AVR_VGA_TILE_MODE3_V010.s|h 144x120, 18x15, 8x8 tiles resolution VGA driver tile engine with 6 sprites supporting 2 full transparent, 1 opaque with scaling, 3 opaque, no scaling. Supports sprite clipping left, right, bottom, top NOT supported, must be done by caller.

As you can see there are (6) NTSC tile engines and (3) VGA tile engines. The reason there are more NTSC engines is that with NTSC we have more time per pixel for calculations. VGA is a tighter timing model and hard to draw graphics with the AVR 8-bit as well as the NTSC.

NOTE	The DVD ROM source might have more engines on it that shown here since we are constantly developing the sources and API. These engines were what was available at the time of writing of this manual. Therefore, the DVD sources directory is always the most up to date, so always check it out and read all the README files for last minute additions and changes.

Similar to the bitmap drivers, the tile map drivers are all very similar in design and architecture. If you understand one of them you understand all of them. The only thing that changes is maybe the resolution of the tile map, the size of the tile bitmaps themselves, the sprite support etc. However, the data structures, names and fundamental method that you set a tile mapped mode up for each driver is the same. With that in mind, let's use the first tile map driver in the list as a reference example; ***XGS_AVR_NTSC_TILE_MODE1_V010.s***, but everything we discuss will crossover to other drivers as well. This driver has the specs of 160x192 with 8x8 pixel tiles, therefore the tile map is 20x24. Figure 16.17 illustrates this tile mode and annotates important details.

Figure 16.17 – A tile map mode illustrated.



Referring to Figure 16.17, we see that the primary data structure that holds the ***"tile map"*** itself is a 2D array of bytes, 20x24. Each byte represents an index into the ***"tile bitmaps"*** themselves. The pointer to the tile map is always called ***tile_map_ptr*** and is a 16-bit pointer declared in the master graphics library module **XGS_AVR_GFX_LIB_V010.c** along with various other tile map related values, here's a glimpse:

```
volatile UCHAR *tile_map_ptr       = NULL;
volatile UCHAR *tile_bitmaps_ptr   = NULL;
```

tile_map_ptr is very key, you need to assign it to the tile map that you want displayed. This must be at least 20x24 bytes and located in SRAM. For example, you might declare the data as follows (excerpted from a demo):

```
// a example tile map 20x24 declaration
volatile UCHAR tile_map[20 * 24] =
{
//  Column
//  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19

    1, 0, 0, 0, 0, 0, 0, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,    // row 0
    0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 1
    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 2
    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 3
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 4
    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 5
    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 6
    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 7
    0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0,    // row 8
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 9
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 10
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 11
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 12
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 13
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 14
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 15
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 16
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 17
```

```
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 18
    2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3,    // row 19
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,    // row 20
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 21
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 22
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 23
};
```

As you can see the tile map is mostly 0's this means that tile index 0 which represents tile bitmap 0 will be displayed in all those locations. Furthermore, tile bitmaps 1,2, and 3 are used in the map. So as it stands we would need only 4 tile bitmaps for this particular tile map. Now, before, we move on, I want to make a comment and let it sink in for later. What if we made this tile map 2x as high? That is 48 rows instead of 24 rows? Well, then we could point the ***tile_map_ptr*** lower down in the data structure and the driver would still draw it. In other words, let's say that we did make the tile map 20x48, so its 2x as high as usual. Then we make the assignment of the ***tile_map_ptr*** to the data structure like this:

```
tile_map_ptr = tile_map;
```

Then we would see the top of the tile map, the 20x24 tiles represented by the data. But, what if we moved the pointer a little bit? For example, added the row width of 20 bytes to it like this:

```
tile_map_ptr += 20;
```

In other words we are adjusting the pointer by the ***"memory pitch"*** of the tile mode:

```
tile_map_ptr += TILE_MAP_WIDTH;
```

This would have the effect of ***"scrolling"*** the image down! Similarly, if we made the tile map wider than its supposed to be then we can achieve horizontal scrolling. But, the engine needs to be told the new memory pitch for this to work. More on scrolling later, but for now, I just want you to realize both the tile map and the tile bitmaps are just memory regions, you can put anything you want in them and you can move the pointers ***tile_map_ptr*** and ***tile_bitmaps_ptr*** around as you wish. The later pointer we need to discuss a little more.

The second variable in the previous pointer declaration listing ***tile_bitmaps_ptr*** points to the actual tile bitmaps you want uses in the tile map. So in your code you will typically define a section of SRAM memory to store the tile bitmaps. In this case, the tile bitmaps are 1-byte per pixel, 8x8, thus each tile bitmap takes 64 bytes of memory. Here's an excerpt from one of the demos that declares NTSC tile bitmap:

```
volatile UCHAR tile_bitmaps[64] =
{
    NTSC_RED,NTSC_RED,  NTSC_RED,  NTSC_RED,  NTSC_RED,  NTSC_RED,  NTSC_RED,  NTSC_RED,
    NTSC_RED,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_RED,
    NTSC_RED,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_RED,
    NTSC_RED,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_RED,
    NTSC_RED,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_RED,
    NTSC_RED,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_RED,
    NTSC_RED,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_BLACK,NTSC_RED,
    NTSC_RED,NTSC_RED,  NTSC_RED,  NTSC_RED,  NTSC_RED,  NTSC_RED,  NTSC_RED,  NTSC_RED,
};
```

Although, it's a bit hard to see, this code declares a "box" shape. The use of named constants in the code makes the image a bit hard to make out; however, here's another example from another demo that uses VGA color values directly encodes in RGB that's a bit easier to make out:

```
volatile UCHAR tile_bitmaps[NUM_TILE_BITMAPS * TILE_BITMAP_SIZE + 1 ] =
{
    3,3,3,3,3,3,3,3,
```

```
    3,0,0,0,0,0,0,3,
    3,0,0,0,0,0,0,3,
    3,0,0,0,0,0,0,3,
    3,0,0,0,0,0,0,3,
    3,0,0,0,0,0,0,3,
    3,0,0,0,0,0,0,3,
    3,3,3,3,3,3,3,3,
};
```

You can clearly make the image out. In this case the inside would be RGB (0,0,0) and the outside border is RGB (0,0,3) which is pure **RED**.

As a side note, the bitmap definitions of NTSC and VGA are about all that's different from a programming point of view for the different drivers. This is the beauty of the level of abstraction the drivers have. You code everything the same, but only the ***"data"*** representing the bitmaps is different. Moreover, this isn't really a problem since once you start developing any applications with size you will use a tool that outputs either NTSC or VGA formatted bitmaps that you can import directly as C code, data statement or read right off SD.

TIP	Remember the VGA encoding is RGB, but the bits are LSB to MSB so the upper 2-bits are always 0's then the actually RGB values are in BGR format left to right (0,0, B1B0, G1G0, R1R0), thus 11 in the lowest bit position creates a pure RED.

Now, the above example only declare the data storage of the bitmaps, they don't point the global pointer tile_bitmaps_ptr to the storage, thus in all your programs you would do something like:

```
tile_bitmaps_ptr = tile_bitmaps;
```

This step in addition to the ***tile_map*** pointer assignment are two steps that are needed to send the tile engine driver the tile map data and bitmap data at very least. At this point, hopefully, you see how easy tile mapped graphics are. The only steps you need to perform in general are:

- Declare a tile map either the same size as the mode or larger to support scrolling and fill it full of tile indexes.
- Declare the tile bitmaps and place byte data representing the pixels of the 8x8 (usually) bitmaps.
- Assign the pointers ***tile_map_ptr*** and ***tile_bitmaps_ptr*** respectively to your data structures.

These are the main steps you need and you will immediately see data on the screen. Of course there are some more details we need to discuss, but if you understand these steps you are 99% there to using the tile mapped modes.

16.3.1 Deconstructing the Tile Map ASM Driver and the Header File

Since there are so many tile map drivers, we can't cover them all, so we are going to stick with the plan and cover the tile engine driver that we have been discussing **XGS_AVR_NTSC_TILE_MODE1_V010.s|h**. The other tile map engines both NTSC and VGA are very similar, the only additions will be in relation to ***sprite support*** and we will cover that as a separate subject shortly. Now, since ASM programming is different than C programming obviously, the ASM file acts as a bit of a header as well, so we are going to explore some elements from both the ASM file and header file. Moreover, ASM programs don't usually have ***"header"*** files, they have ***"includes"***, but since we are trying to tightly integrate ASM and C and make things easy to understand we have used the convention to call the external include files for the ASM files headers as well and use the .h extension just to make things easy.

Let's begin with the entire **"Data section"** from the ASM driver, it contains some extra internal declarations, but we can parse them out of our discussions:

```
////////////////////////////////////////////////////////////////////////////////////////////////
// LOCAL INITIALIZED VARIABLES (SRAM)
////////////////////////////////////////////////////////////////////////////////////////////////

.section .data

// tile map variables
tile_index_x:             .byte 0
tile_index_y:             .byte 0
tile_bitmap_row:          .byte 0

tile_map_save_ptr:        .byte 0,0
tile_bitmap_base_ptr:     .byte 0,0

// current video line
curr_line:                .byte 0,0

// current raster line 0..virtual rez-1
curr_raster_line:         .byte 0
raster_line:              .byte 0

// amount of vertical scroll 0..7 (caller controlled)
tile_map_vscroll:         .byte 0

// the logical tile map width, used to access rows, this support horizontal course scrolling,
(caller controlled)
tile_map_logical_width:   .byte  TILE_MAP_WIDTH

// tile bitmaps pointer array buffer, 20 tiles, 16-bit pointer for each that points to a bitmap
row to render.
// 2x20 = 40 bytes
tile_bitmaps_ptr_array:   .byte 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
                                0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0

                          .byte 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
                                0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0
```

The majority of these variables are internal to the driver, but they are interesting to take a look at. I have highlighted a couple variables that actually exported from the driver, they are:

tile_map_vscroll — This ranges from 0..7 and is the vertical scroll amount to shift the display to support vertical ***"fine scrolling"***.

tile_map_logical_width — This is the "logical width" of the tile map, even though the visible display is always 20x24 in this driver, this variable indicates to the driver the "virtual row pitch" to support horizontal ***"course scrolling"***.

We touched on scrolling support and we are going to talk about in the next section, but these variables above are directly involved in the interface to the driver to support this feature. Next, let's take a look at what's called the **"Externals section"** of the driver:

```
.section .text

////////////////////////////////////////////////////////////////////////////////////////////////
// EXTERNALS
////////////////////////////////////////////////////////////////////////////////////////////////

.extern tile_map_ptr      // 16-bit pointer to tile map
.extern tile_bitmaps_ptr  // 16-bit pointer to 8x8 tile bitmaps

////////////////////////////////////////////////////////////////////////////////////////////////
// GLOBALS EXPORTS
////////////////////////////////////////////////////////////////////////////////////////////////

// export these to caller, so he can interrogate and change
.global curr_line                 // read only, current physical line
.global curr_raster_line          // read only, current logical raster line
.global tile_map_logical_width    // read/write, logical width of tile map
.global tile_map_vscroll          // read/write, current vertical scroll value 0..7
```

The driver imports ***tile_map_ptr***, and ***tile_bitmaps_ptr*** then the driver exports the remaining variables. All this means is that the driver expects another module (C or ASM) to declare the tile pointers, but the other variables the driver declares and external modules can link with them and use them. Again notice the ***curr_line*** and ***curr_raster_line*** variables, these are once again the same variables used in the bitmap drivers and are very important to determine where the video raster is and what it is doing. Remember, that we don't want to disturb the video buffers while the raster is in the active part of the scan. Ok, so that's some of the ASM file itself, now let's take a look at the header for the tile engine, it has a lot of interesting constants and macros that you can take advantage of as well as use in the other tile map engines since they are all more or less the same other then resolution. The header **XGS_AVR_NTSC_TILE_MODE1_V010.h** starts off with a very important declaration for the interrupt driver:

```
// interrupt rates for NTSC/VGA
#define VGA_INTERRUPT_RATE   910
#define NTSC_INTERRUPT_RATE  1817
#define VIDEO_INTERRUPT_RATE (NTSC_INTERRUPT_RATE)
```

Notice how both interrupt rates are declared, but then the NTSC rate is bound to the 3rd constant ***VIDEO_INTERRUPT_RATE***, this is how the video rate from the driver is "passed" to the final driver initialization code. This constant is needed to setup the timer interrupt that calls the video driver (whatever driver that may be). The driver always needs to be called at the rate of the horizontal scan rate for the video driver. In the case of NTSC this is **15.734 KHz**, and for VGA this is roughly 2x that rate or **31.4 KHz**. Both NTSC and VGA monitors are tolerant of these rates being slightly different (especially VGA since they support many rates), but you have to be careful that you drive the signal such that its clean and without noise. In any event, these counter constants are plugged into the various confusing timer interrupt values with clock dividers etc. to finally generate the interrupts that call the video driver.

Next up in the header are some of the constants for NTSC signal levels:

```
// with the current D/A converter range 0 .. 15, 1 = 62mV
#define NTSC_SYNC           (0x0F)     // 0V
#define NTSC_BLACK          (0x4F)     // 0.3V (about 30-40 IRE)

#define NTSC_CBURST0        (0x40)
#define NTSC_CBURST1        (0x41)
#define NTSC_CBURST2        (0x42)

#define COLOR_LUMA    (NTSC_CBURST0 + 0x40)

#define NTSC_GREEN    (COLOR_LUMA + 0)
#define NTSC_YELLOW   (COLOR_LUMA + 2)
#define NTSC_ORANGE   (COLOR_LUMA + 4)
#define NTSC_RED      (COLOR_LUMA + 5)
#define NTSC_VIOLET   (COLOR_LUMA + 9)
#define NTSC_PURPLE   (COLOR_LUMA + 11)
#define NTSC_BLUE     (COLOR_LUMA + 14)

#define NTSC_GRAY0    (NTSC_BLACK + 0x10)
#define NTSC_GRAY1    (NTSC_BLACK + 0x20)
#define NTSC_GRAY2    (NTSC_BLACK + 0x30)
#define NTSC_GRAY3    (NTSC_BLACK + 0x40)
#define NTSC_GRAY4    (NTSC_BLACK + 0x50)
#define NTSC_GRAY5    (NTSC_BLACK + 0x60)
#define NTSC_GRAY6    (NTSC_BLACK + 0x70)
#define NTSC_GRAY7    (NTSC_BLACK + 0x80)
#define NTSC_GRAY8    (NTSC_BLACK + 0x90)
#define NTSC_GRAY9    (NTSC_BLACK + 0xA0)
#define NTSC_WHITE    (NTSC_BLACK + 0xB0)
```

These constants simply make it easy to write common colors down if you are designing bitmaps by hand for the tile engine and/or are easy to remember during testing. A couple things to remember; first all colors are 8-bit and in the format: **[LUMA3..0:CHROMA3..0]**. Furthermore, color **15** or **0x0F** in hex is the **"absence"** of color or disables the color burst signal. Use this value for really clean b/w or grayscale imagery or if you don't want to use color at all.

The next declaration in the header define sizes of things, widths, heights etc.

```
// width and height of screen
#define NTSC_SCREEN_WIDTH     160
#define NTSC_SCREEN_HEIGHT    192

#define SCREEN_WIDTH          160
#define SCREEN_HEIGHT         192

// controls the number of lines/pixels to shift the display vertically/horizontally,
// later assign to global variable, so caller can control
#define NTSC_VERTICAL_OFFSET   36
#define NTSC_HORIZONTAL_OFFSET 80

#define VERTICAL_OFFSET        36
#define HORIZONTAL_OFFSET      80


// these indicate the start, end, of the entire video scan and of the active lines,
// always in terms of real video lines 0
// eg. 0..261 for NTSC, 0..479 for VGA
#define START_VIDEO_SCAN      0
#define START_ACTIVE_SCAN     (VERTICAL_OFFSET)
#define END_ACTIVE_SCAN       (SCREEN_HEIGHT + VERTICAL_OFFSET+1)
#define END_VIDEO_SCAN        261
```

Typically, you won't want to modify these, but you can fudge the horizontal and vertical offset constants if you want to ***shift*** the image on the screen around a bit on the TV. These control the amount of horizontal delay and blank lines leading the display image. Moving on, the next bit of constants are actually very important macros:

```
// simply interrogate the video driver's curr_line variable and determine where the scanline is
// returns true/false boolean
#define VIDEO_ACTIVE(video_line)          ( ((video_line) >= START_ACTIVE_SCAN && (video_line) <=
                                          END_ACTIVE_SCAN) ? 1 : 0)

#define VIDEO_INACTIVE(video_line)        ( ((video_line) < START_ACTIVE_SCAN || (video_line) >
END_ACTIVE_SCAN)
                                          ? 1 : 0)

#define VIDEO_TOP_OVERSCAN(video_line)    ( ((video_line) >= 0 && (video_line) <
START_ACTIVE_SCAN) ? 1 : 0)
#define VIDEO_BOT_OVERSCAN(video_line)    ( ((video_line) > END_ACTIVE_SCAN) ? 1 : 0)
```

These ugly macros using ternary conditions let you interrogate the raster line and determine if its in top overscan, active region or bottom overscan. The macros rely on the previous definitions and you pass them the variable you want tested which will typically be ***curr_line*** from your application (exported from the driver itself), but this need not be the case, you might want to test another variable thus the macros support a test parameter. Lastly, the header has some convenient constants which you can always count on in all the drivers, so you can make your data declarations, and conditionals clear and not full of specific values:

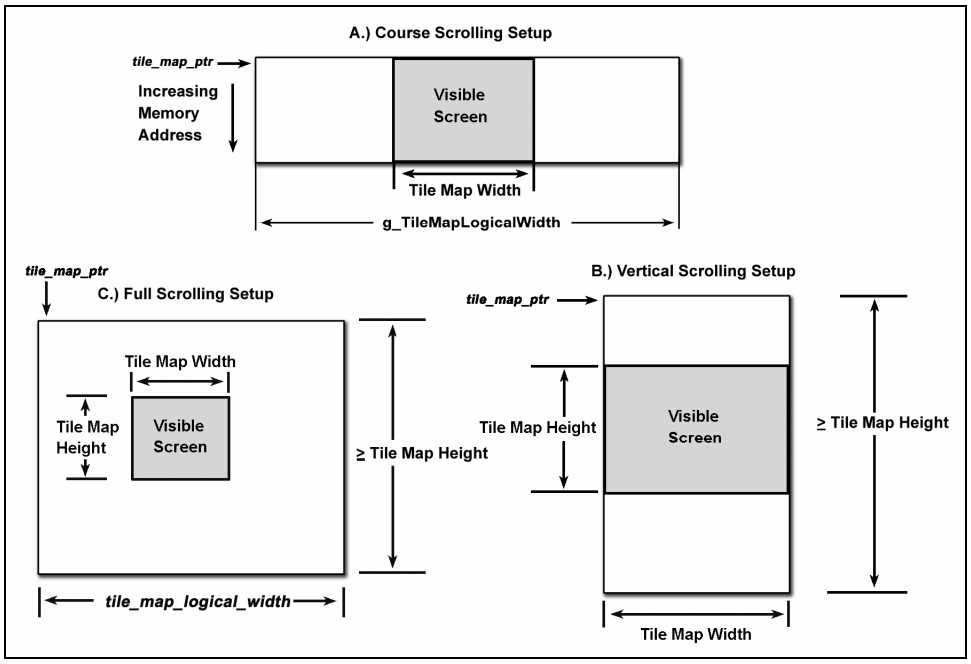
```
// number of bits each pixel is encoded with
#define NTSC_PIXELS_PER_BYTE   4
#define PIXELS_PER_BYTE        4

// number of clocks per pixel (8 in the case of NTSC)
#define CLOCKS_PER_PIXEL       8

// tile engine defines
#define TILE_MAP_WIDTH         20
#define TILE_MAP_HEIGHT        24

// tile engine bitmap constants
#define TILE_BITMAP_WIDTH      8   // width and height of tile bitmap
#define TILE_BITMAP_HEIGHT     8
#define TILE_BITMAP_SIZE       64  // size of tile bitmap
```

For example, no matter what driver you use, if you want to know the width of the tile map, you can use the constant ***TILE_MAP_WIDTH*** in your code and so forth.

16.3.2 Scrolling Tile Maps

In the previous discussions we have touched up scrolling a few times and even indicated the variables you need to interface with to support this feature. Scrolling with tile engines is a very important feature and can be complex to support if you don't approach the driver design properly. For example, since we use a pointer to the tile map we can point it anywhere in memory and more or less support ***"course scrolling"*** on a tile by tile basis very easily by adjusting the ***tile_map_ptr*** by the tile map row pitch up or down, or by changing the address by 1 byte which would effectively horizontally course scroll the image. This is where the ***tile_map_logical_width*** variable comes in. Normally, you would set ***tile_map_logical_width*** to the tile map width itself ***TILE_MAP_WIDTH***, but say you wanted to have 10 screens of data horizontally then you would set the variable to ***10*TILE_MAP_WIDTH***. The tile driver then monitors this variable and then uses it in its addressing of data, instead of incrementing the row pointer by 20, for a 20x24 tile map, it increments it by ***tile_map_logical_width***, and then for all intent purposes you have horizontal course scrolling. Figure 16.18 illustrates both setups for horizontal and vertical course scrolling.

Figure 16.18 – Course scrolling for horizontal and vertical setups.



Thus to support vertical course scrolling you need to add rows to your tile map, set the logical width to the standard size of the mode and then adjust the ***tile_map_ptr*** by the memory pitch per row to scroll up/down. For example, the pitch would be 20 for a 20x24 tile mapped mode, we have seen this in examples above. Now, for the horizontal scrolling you need to not only define the tile map properly, but you need to adjust the ***tile_map_logical_width*** variable, so the driver knows how wide you want to interpret the tile map data as. Then you once again adjust the ***tile_map_ptr***, but this time by +-1 if you want to scroll left right. Of course, you can't scroll into addresses that don't exist, so be careful of bounds checking.

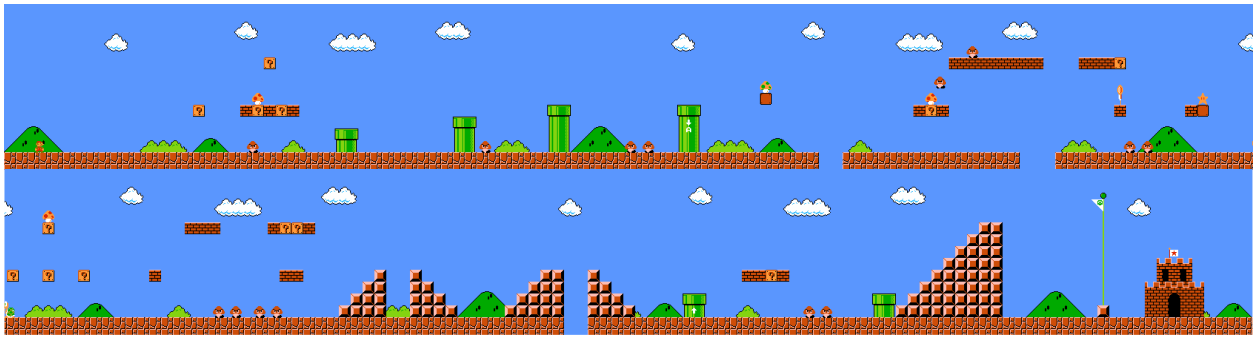
As an example, of what a tile map that is say 30x24 that you want to display with a 20x24 tile engine, you would declare it like this:

```
// the tile map 30x24, but the tile map engine is only 20x24, thus you can scroll right/left 10 tiles
volatile UCHAR tile_map[ 30*24 ] =
{

//  Column
//  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

    1, 0, 0, 0, 0, 0, 0, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,    // row 0
    0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 1
    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 2
    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 3
```

```
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 4
    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 5
    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 6
    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 7
    0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 8
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 9
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 10
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 11
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 12
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 13
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 14
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 15
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 16
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 17
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 18
    2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3,    // row 19
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,    // row 20
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 21
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 22
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,    // row 23
};
```

As an example of how far you can take this, take a look at Figure 16.19 below, it shows the entire game world layout (stacked on each other to fit on the page better). But, this is totally horizontally scrolled during game play as you know if you have played.

Figure 16.19 – Super Mario Brothers has a whole lot of screen data which is horizontally scrolled.



Fine Scrolling

Let's finish off with the concept of ***"fine scrolling"***. This means you can scroll the tile map image by a single pixel or line, horizontally or vertically respectively. This is much harder to since it usually means buffering the video image, and shifting it, also, you have to have a tile map that is at least (WIDTH+1 x HEIGHT+1) in size. The tile engines we have developed only support vertical fine scrolling at this point. In the future, I might redo a couple and add horizontal fine scrolling, but I was trying to keep the ASM understandable and the more features added and optimizations used make the code really hard to reverse engineer. In any event, fine scrolling works via the fine scroll variable called ***tile_map_vscroll***. All you have to do is write values 0..7 to this variable from your application and the tile image will scroll vertically. But, once you hit 7 then it rolls back around to 0. The solution to achieve continuous smooth vertical scrolling then in the following algorithm:

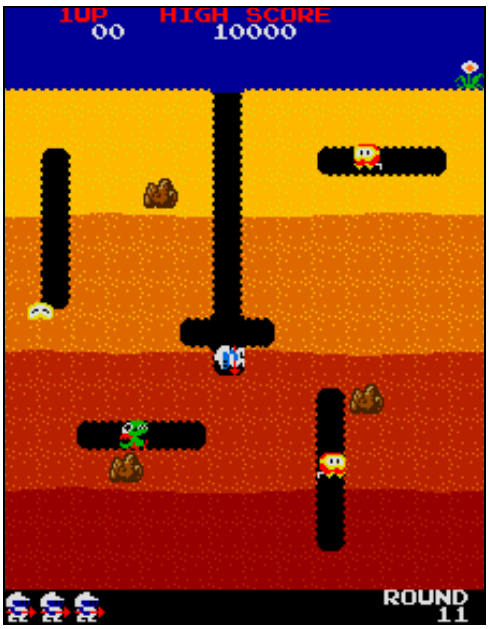
```
if (++tile_map_vscroll > 7)
   {
   // reset fine scroll
   tile_map_vscroll = 0;

   // course scroll one line
   tile_map_ptr += tile_map_logical_width;

   } // end if
```

Notice I used ***tile_map_logical_width*** instead of ***TILE_MAP_WIDTH***, this is a good tactic since this is always setup (you set it) to the actual logical memory pitch of your tile maps. The other direction of vertical scrolling would be the same, you subtract and then adjust the course pointer up a line.

16.3.3 Animating Tile Maps

At this point you should have a good idea how to create tile maps, bitmap data, and scroll the tile map itself. The last piece of the puzzle is ***"animating"*** the tile map itself. In other words, moving things around and changing the tile indexes.

Figure 16.20 – The Atari classic game "Dig Dug".



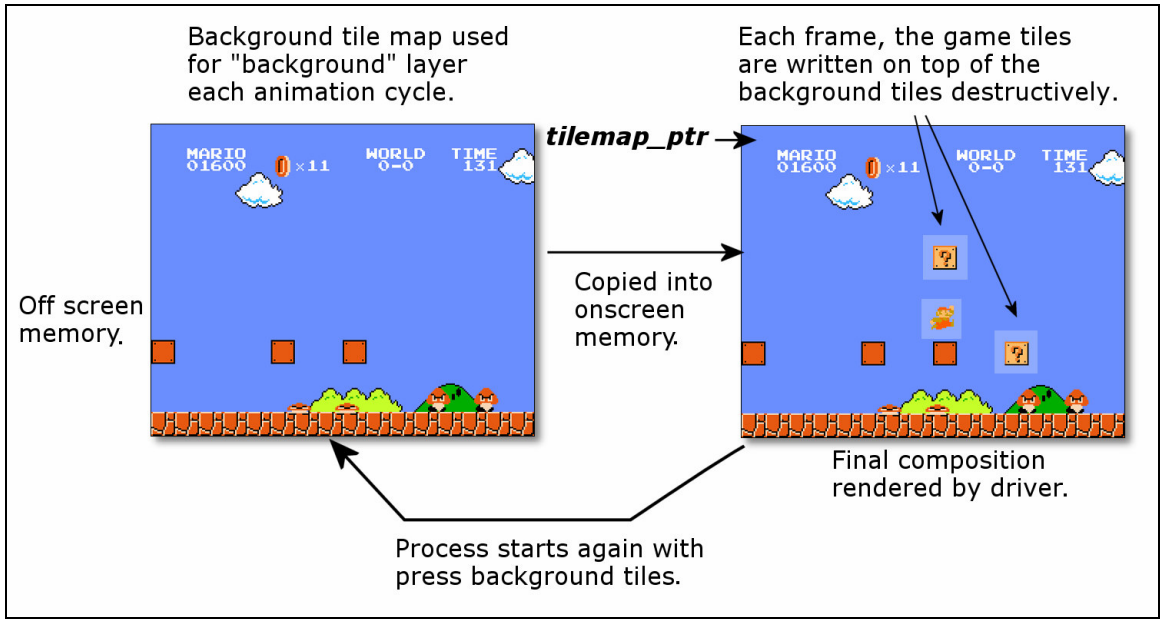
For example, say that you had written a game like Atari **"Dig Dug"** show in Figure 16.20. This game is tile mapped of course and the characters are sprites. All the dirt, rocks, flowers, text, etc. all tiles. Now, say that the like "Dig Dug" character (guy in the white suit) is digging his way down as he is in the screen shot and you want to replace the current tile he is on with the ***"blank"*** or background tile, this is a form of animation. The code might look like this:

```
tile_map[ dig_tile_x + dig_tile_y*TILE_MAP_WIDTH ] = background_tile;
```

So to animate any tile you simply write into the tile with the new bitmap index that you want displayed. Now, there are some problems with this. First, when you write into the tile map, whatever you overwrite is gone, thus if you want to restore it later you need to remember it. Secondly, you can only place new tiles on tile map boundaries, thus, you can smoothly move tiles (this is what sprites are for), hence, tile animation for motion is very jerky since you have to move things a whole tile at a time. There are ways around this by using **"sub-tile"** animation techniques, which I will touch on in a moment.

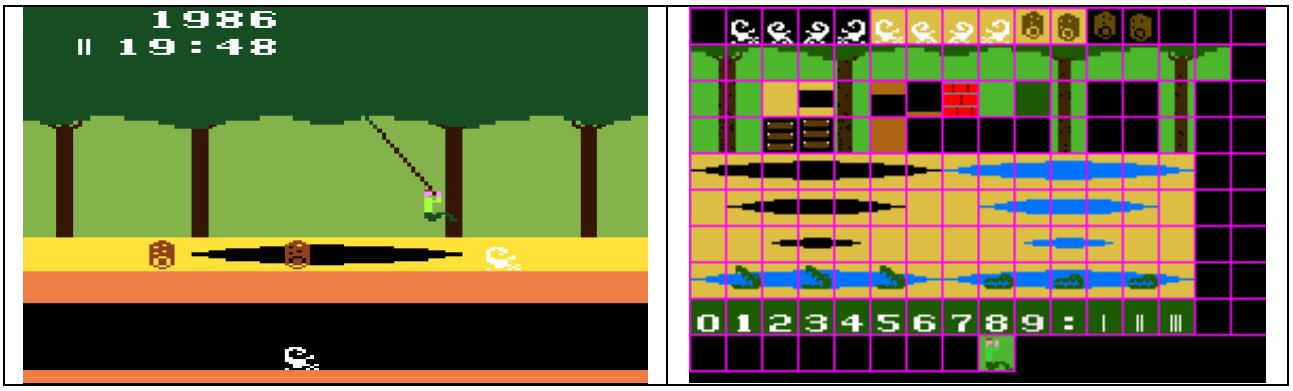
Let's address the first problem; the loss of data and potential storage of the data. There are a couple ways to approach this problem. One method is to build a new tile map every single frame of animation. That is every 30th or 60th of second (whatever frame rate you run the animation at), you actually copy the entire original unscathed tile map data to another ***"working tile map"***. This working tile map you then overwrite, damage, change tiles, and you need not remember anything since you are going to refresh this next frame once again. Figure 16.21 shows this graphically.

Figure 16.21 – Copying the background tile map into a working tile map, so you can overwrite it.



This is fine, but you double the amount of tile map memory you need. However, many times this is much better than trying to remember every tile that you draw onto in your animation loop. For example, coming back to the previous Atari "Pitfall" example, say you want to move the scorpion across the screen as shown in Figure 16.22.

Figure 16.22 – Tile animating the "Pitfall" scorpion.



The left image shows the tile map we want to move the scorpion in (the top scorpion) and the tile bitmaps again. As you can see, there are 2 frames of animation in the tile map (for right/left and the different potential background colors), but as we move the scorpion and animate it, we are going to either leave a trail of destruction or incorrect tiles. That is as you move the scorpion from left to right, you must replace what tile was under it, therefore you either need to keep track of it, or you need to use the above aforementioned ***"working copy"*** of the tile map. However, for the fun of it let's look at what the code might look like to just remember what was under the scorpion and replace that every time we move it. Let's assume that the scorpion tile bitmap frames are numbered 5,6 that we want to animate (right walk as shown in the figure if you count tile bitmaps from left to right, top to bottom starting from 0), and we aren't going to worry about timing issues, borders, collision, just the animation and the background tile copying. The code would look something like this:

```
// setup, first store the tile under the scorpion before entering the loop
tile_under_scrorpion = tile_ptr[ scorp_x + scorp_y*TILE_MAP_WIDTH];

// initialize animation frame
scorpion_frame = 5;

// main animation loop
while(1)
{
// replace what's was under the scorpion
tile_ptr[ scorp_x + scorp_y*TILE_MAP_WIDTH] = tile_under_scrorpion;

// move the scorpion
scorp_x++;

// store whats under scorpion in new position
tile_under_scrorpion = tile_ptr[ scorp_x + scorp_y*TILE_MAP_WIDTH];

// now overwrite that tile with the scorpion
tile_ptr[ scorp_x + scorp_y*TILE_MAP_WIDTH] = scorpion_frame;

// animate scorpion walking animation
if (++scorpion_frame > 6)
   scorpion_frame = 5;

} // end while
```

This is obviously a very crude example, and a lot of details are missing, but the idea here is clear I think, you need to constantly scan the tile index under the scorpion before overwriting it then you need to replace it when the scorpion moves. This is the basis of animation:

"Erase – Move – Draw"

This pattern is performed in one way or another to achieve animation in all cases. You might erase the whole screen, a portion, etc. then you need to move and animate the objects, then you need to draw them and present them to the user.

16.3.4 Tile Engines and Sprite Support

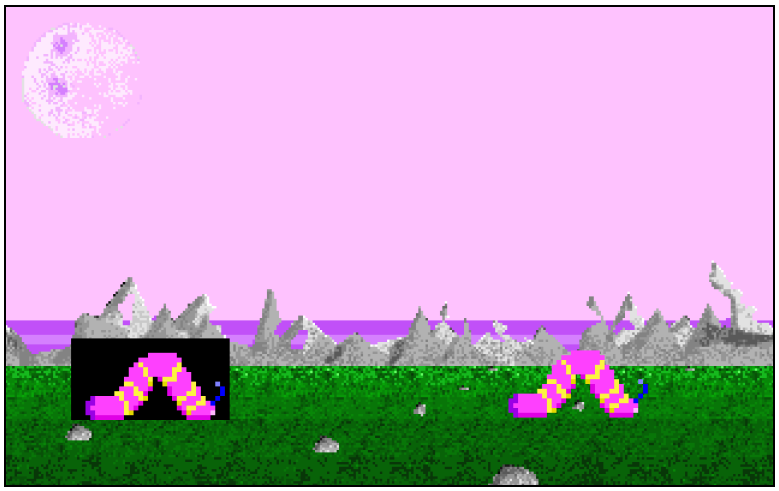
Sprites as mentioned before augment tile engines by allowing a fixed number of ***"overlaid"*** bitmaps to move freely around the tile map display without disturbing the tile map itself. Sprites are typically draw after the tile map or on top of it either with software algorithms or physical hardware signal mixing. In our case, we do it all with software. Sprite support is added by generating each video line of the display with the tile map information then overwriting it as it goes out to the raster with the sprite data. This takes a lot of computation; masking, OR'ing, AND'ing, lookups etc. so it's hard to support a lot of sprites since you have to do all the calculations in such little time. But, the tile engines support anywhere from 3 to 6 sprites with the sprites being pretty straight forward bitmaps to supporting scaling operations. In this section, we are going to discuss the sprite support and the key data structures needed to get them up and running. Again, the idea of the manual is more of a hands on practical reference, so you will see many solid examples in the demo section of the manual. The idea here is to simply give you an overview and the vocabulary to understand the various features of the sprites in the XGS AVR tile engine drivers.

First, just to make sure we get what a sprite is again, take a look at the Atari "Dig Dug" screen shot in Figure 15.21, the background is made up of tiles, but the foreground objects like the player and all the game characters are sprites. Notice how the overlap the tiles and can move around anywhere on the screen? For example, on the left edge of the screen you see a little "Pooka" in Google mode running right thru the solid rock. If it wasn't a sprite, this would very hard to accomplish, but with a sprite its trivial, the sprite is drawn on top of the tiles and you can typically place the sprite anywhere on the screen simply by positioning its x,y coordinate. Moreover, sprites typically have transparency support and scaling.

Sprite Transparency

Sprite transparency allows you to use a single color code (0 in our case) to represent "transparent" or see thru. Therefore, any pixel of the bitmap that represents the sprite that is drawn with the transparent color will allow you to see the background. Figure 16.23 shows and example of transparency and opaque background.

Figure 16.23 – Transparent and non-transparent sprite rendering.



As you can see, the transparent sprite on the right is much more what you would expect. However, the transparency comes at a price – there is a lot more calculation for the sprite engine since it has to test each pixel value for transparency and then if transparent draw the background tile map data, else draw the sprite bitmap. Therefore, sprites without transparency have their place as well. For example, if you are placing a sprite on a background with a constant background color (like the sky in this example), you could use a non-transparent sprite, but simply set its background color (black in this case) to the background color and that big rectangular region of opacity in the non-transparent version wouldn't be noticeable. Another good example is a "space game". In games, where the background is primarily black, you don't need or care about transparency.

In any event, transparency costs computation, so the sprite engines have limitations with this feature that we will get to shortly.

Scaling

One on the more powerful features that many sprite engines have (since they are hardware based in many cases) is that the sprites can be scaled on the fly before rendering. In other words, you may define a sprite as only a 8x8 or 16x16 bitmap, but when its drawn you can set the "scale" of the sprite to 1x,2x,4x, etc. however the particular sprite engine works. In our case, the scaling feature (in the drivers that support it) only support 2x scaling on the x axis. In other words, when you enable scaling the sprite will be 2x as wide. This is accomplished by doubling each pixel as its drawn. Thus, you don't need to supply any more data, just flip the scaling on in the sprite data structure (which we will get to) and the sprite is drawn get 2x as wide.

Working with Sprites

First, sprites are supported only in tile map engines. You could add sprites to a bitmap engine in your own design, but historically sprites are added to tile map engines to overcome the constraints of the tile maps themselves. Also, its very important that you realize that each sprite engine might have slight differences in the number of sprites and the features that each sprite supports. The only way to figure this out is to open up the source code for the tile map driver and read the information at the top of the file. Each driver clearly explains the number of sprites and what they support. The good news is the ***"sprite interface"*** from a programmer's point of view is ***always*** the same no matter if you are using NTSC or VGA graphics. Sprites are always defined by the same data structure (shown in a moment) and manipulated in the same way. The thing that changes from driver to driver and NTSC to VGA are the special

feature bits and of course the bitmap definition of the sprites. With that in mind, let's discuss how sprites are used in the tile map engines. And this time let's start with a VGA example since we have been using NTSC examples so much.

For our discussion we will use the driver **XGS_AVR_VGA_TILE_MODE2_V010.s**, this is a VGA tile driver with the following specs:

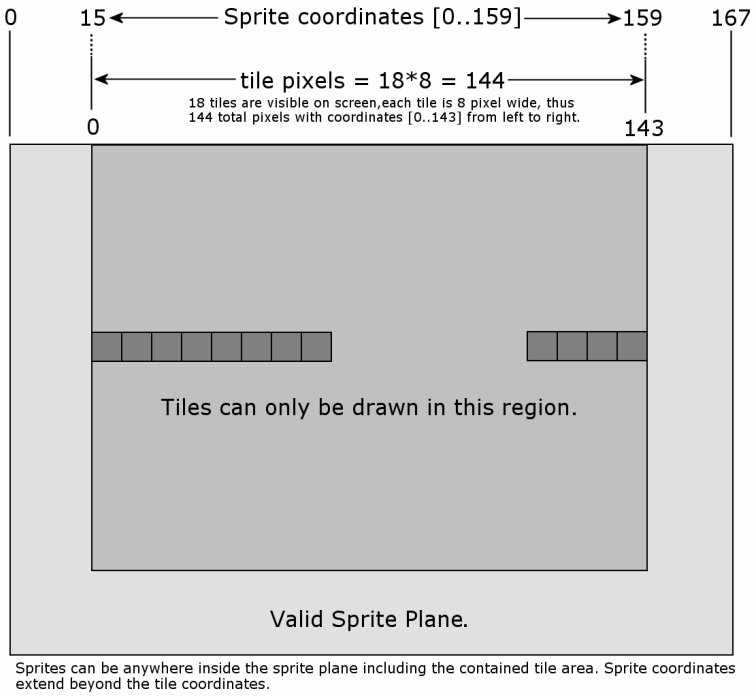
- 144x120 pixels resolution.
- 18x15 tiles each 8x8 in size.
- 3 sprites; 2 with transparency, 1 that's opaque, but supports 2X horizontal scaling.

The engine supports (3) 8x8 pixel sprites (same size as tile bitmaps) with full 6-bit color (the upper 2-bits of every VGA color byte are sync bits and unused). The sprites have some limitations;

- Sprites 0,1 are support full transparency (via color key value for the pixel data)
- Sprites 2 is opaque (value 0 will show black not transparent); however, sprite 2 has 2X scaling.

All sprites support scrolling off the right, left, and bottom of the screen. The sprite coordinates are slightly translated relative to the tile coordinates, so sprite x = 0, places the sprite just beyond the left edge of the screen, similarly setting a sprite's x = 159 places the sprite beyond the right edge of the screen. In other words, the sprite coordinates look like the diagram shown in Figure 16.24.

Figure 16.24 - Sprite coordinates relative to the tile map.



Sprites can be anywhere inside the sprite plane including the contained tile area. Sprite coordinates extend beyond the tile coordinates.

It's a nice feature to allow sprites to be drawn off screen, so they can ***"smoothly"*** enter and exit the game play field. The interface to the tile engine from the C/C++ caller consists of a set of pointers to data structures as well as some read/write variables exported from the tile engine that the caller needs to control the engines operation. Here are the important variables that the tile engine must have initialized:

```
.extern tile_map_ptr      // 16-bit pointer to tile map, allows indirection
.extern tile_bitmaps_ptr  // 16-bit pointer to 8x8 tile bitmaps, allows indirection
.extern sprite_table      // sprite table base address containing sprite records, must be static
data structure
```

First, are the tile map, tile bitmap pointers which we have see before, and finally the sprite data structure itself. To review, the tile map is a simple byte array m x n, where each byte represents the tile ***"index"*** to be rendered at that position. Each index refers to the tile bitmap that should be drawn. The tiles themselves are a contiguous array of 8x8 pixel (byte) tiles, thus 64-bytes per tile (encoded in either NTSC or VGA formats). Both of these data structures are referred to by pointers to allow movement of the data structure in real-time. So the caller would typically dynamically or statically allocate the tile map and tile bitmaps then point the below pointers at them, so the tile engine could locate them.

Now, the new data structure is the sprite table base address, this is a static name, thus the caller will typically define the ***sprite_table*** as a static array of sprite records, where each record is a of the type:

```
  // SPRITE type, 8 bytes each
  typedef struct SPRITE_TYP
      {
      // SPRITE members, note depending on driver some are used, some are not
      UCHAR state;    // state of sprite
      UCHAR attr;     // bit encoded attributes
      UCHAR color;    // general color information
      UCHAR height;   // height of sprite
      UCHAR x;        // x position of sprite
      UCHAR y;        // y position of sprite
      UCHAR *bitmap;  // pointer to bitmap (2 bytes)
      } SPRITE, *SPRITE_PTR;
```

NOTE	The ***SPRITE_TYP*** typedef is declared in the GFX header which we will discuss in the next main section.

In most incarnations of the tile engines only attr, height, x, y, and *bitmap are used. The remaining fields are for the caller's own use. Moreover, the attr field "attribute" is rarely used, and simply holds bit encoded attributes of the sprite. Currently, the only attribute that is defined is:

```
#define SPRITE_ATTR_2X       1  // enable 2X scaling mode
```

Which as it states controls sprite scaling on sprite engines that support it (this engine support scaling on sprite 2 only). Other than that the only other interesting field is the bitmap pointer ****bitmap***. This is a pointer that points to the 8x8 matrix of pixels that will be used for the sprite image. The data is in the SAME format as tiles, this you can point it to a tile, or another set of data as you wish. The important thing is that each sprite is 8x8 pixels where each pixel is an VGA (or NTSC) encoded byte. Additionally, value 0 is used for transparent which equates to VGA_BLACK (and ***NTSC_SYNC*** in NTSC encoding which equates to 0 as well), however, the video will not sync when this value is encoded, it directs the renderer not to draw anything, and thus make the pixel transparent to the tile data under it.

Considering the information above, typically you will declare a ***sprite_table***, then fill it up with values, point the bitmap pointers to your sprite information (either tile bitmaps or separate bitmaps) and the driver will take care of the rest. For example, you might define some sprites as follows:

```
// vga sprites
volatile UCHAR sprite_bitmap[ TILE_BITMAP_SIZE*4] =
{

    0,0,0,3,3,0,0,0,
    0,0,0,3,3,0,0,0,
```

```
    0,0,0,3,3,0,0,0,
    0,0,0,3,3,0,0,0,
    3,0,3,3,3,3,0,3,
    3,3,3,3,3,3,3,3,
    3,0,3,3,3,3,0,3,
    0,0,0,3,3,0,0,0,

    0,0,9,9,9,0,0,0,
    0,0,9,9,9,0,0,0,
    0,0,0,9,0,0,0,0,
    0,8,9,9,9,8,0,0,
    0,8,9,9,9,8,0,0,
    0,8,9,0,9,8,0,0,
    0,0,9,0,9,0,0,0,
    0,9,9,0,9,9,0,0,

    48,48,48, 3, 3,48,48,48,
    48,48,48, 3, 3,48,48,48,
    48,48,48, 3, 3,48,48,48,
    48,48,48, 3, 3,48,48,48,
    48,48,48, 3, 3,48,48,48,
    48,48,48, 3, 3,48,48,48,
    48,48,48, 3, 3,48,48,48,
    48,48,48, 3, 3,48,48,48,
    3 ,48, 3, 3, 3, 3,48, 3,
    3 ,48, 3, 3, 3, 3,48, 3,
    3 , 3, 3, 3, 3, 3, 3, 3,
    3 , 3, 3, 3, 3, 3, 3, 3,
    3 ,48, 3, 3, 3, 3,48, 3,
    3 ,48, 3, 3, 3, 3,48, 3,
    48,48,48, 3, 3,48,48,48,
    48,48,48, 3, 3,48,48,48,
};
```

If you blur your eyes then you can see the bitmaps they represent, for example, the first sprite is a little space ship. Also, this definition shows that you can make sprites larger than 8 pixels high.

TIP	All the tile map drivers that support sprites allow the bitmaps to be 8xn where n is from 1 to 32. This is why there is a height field in the sprite type. However, sprites must always be 8 pixels wide currently. Nonetheless, the variable height allows some pretty cool tricks. For example, you can place 2 sprites next to each other horizontally and get a ***"giant"*** 16x32 super sprite!

The next data structure you need is the sprite table itself, and in most cases it will look something just like this:

```
// state, attr, color, height, x,y, *bitmap
volatile SPRITE sprite_table[NUM_SPRITES] = {
                                             {0,0,0,8, 20,30,sprite_bitmap},    // sprite 0
                                             {0,0,0,8, 60,50,sprite_bitmap+64}, // sprite 1
                                             {0,1,0,8, 80,60,sprite_bitmap}     // sprite 2
                                            };
```

In this case, sprite 0 has a height of 8, its (x,y) is (20,30) and the bitmap data starts at ***&sprite_bitmap***, sprite 1 has height 8, (x,y) of (60,50), but this time we offset the base address of the bitmap data to shift it down 64 bytes to the next bitmap, finally sprite 2 is defined in much the same way, but the ***SPRITE_ATTR_2X*** flag is set which will double the width of the sprite when rendered.

In conclusion, sprites add a lot freedom in your tile mapped games and graphics demos since you can move them freely around the screen. The tile engines that support them (both NTSC and VGA) each have limitations, but they are always outlined at the top of the source files. If you are still a little unsure exactly how it all works, don't worry, that's what all the hands on demos are for, to show you concrete examples of everything, not just graphics and sprites.

16.3.5 SRAM Versus FLASH Tile Bitmaps

You might wonder why the tile engines use SRAM for the tile bitmaps rather than FLASH memory? The reason is two fold; first SRAM is much faster than the FLASH memory and easier to deal with in C. The graphics drivers are so

optimized even a single extra clock in the inner pixel loops will be too much. Therefore, SRAM has to be used in most cases. However, FLASH storage can be tolerated for lower resolutions where there is more time per pixel. Nonetheless, the second reason SRAM is better is that you can actually modify the bitmaps in real-time to achieve interesting animation effects. With FLASH you can't do this.

On the other hand, using our precious SRAM to store tile bitmaps that are probably going to be static is terribly wasteful, thus, we would like to store them in FLASH if at all possible, so we free up the SRAM for the tile map memory which ***does*** need to be in SRAM always to support animation. However, if you use static bitmaps and static tile maps, then another approach is to place both the tile map and tile bitmaps into FLASH, but this will really slow down access and the code will have to be much faster. However, if you can squeeze a couple sprites in then you can support truly massive worlds. Of course, another approach is to store the tile maps in FLASH, then cache them in SRAM which gives you the same flexibility, but not the speed hit.

As you can see, graphics is a lot of fun and so is optimizing the code. There are about a billion ways to do things, I have shown you 2-3 of them – so I highly recommend you develop your own graphics drivers that use the SRAM and FLASH in ways that are truly creative and innovative.

16.4 Developing More Advanced Drivers

The drivers we have discussed are just starting points for you to develop your own. Granted they are in assembly language and if you're not an ASM programmer then you are going to have to learn. But, if you are interested in programming microcontrollers then you need to know ASM. The drivers are heavily commented and I tried to keep them as clean as possible and not use many tricks, but due to the poor little AVR 644s limited memory and speed, its really hard to generate bitmap and tile displays without some heavy optimizations and tricks. I suggest that you start off with the simplest tile driver and then see if you can make changes to it. Then once you understand completely how it works you might try developing one yourself. For example, a driver I haven't had time to write is called a ***"super sprite"*** driver. In this kind of driver, we simply support sprites with tiled color backgrounds, that is, no bitmap tile background, just color on tiled basis or scan line basis. This disallows any detailed background, but with 30-40 sprites on the screen you can do a lot of cool space games or other shooters.

End Sample

You've been Terminated…

Well, you made it! If you got this far then you should be a master at the XGS AVR 8-Bit and hopefully see the true power of the Atmel AVR 8-bit series processors. You should be able to apply these techniques to any processor that you might work with in your embedded systems development.

I never get bored playing with these little processors and seeing what they can do and I hope you feel the same! But, with great power comes great responsibility. The techniques learned here can be used for good or evil…

Remember, **SkyNet** started out simple. Nothing more than cyborgs based on 8-bit 6502 processors ruled the planet!

AVRs are 100x more powerful, imagine would kind of cyborg you could develop with that!

Finally, we would love to see what you come up with, so make sure to visit our forums at:

http://www.xgamestation.com/phpbb/index.php

Discuss and show off your demos, games, and applications with other XGS AVR programmers. Also, if you have any questions, comments, or remarks please email us at **support@nurve.net**.