Version 1.0

"Inside the XGS PIC 16-Bit"

User Manual and Programming Guide

Josh Hintze
& Andre' LaMothe

Nurve Networks LLC

XGAMESTATION™ PIC 16-Bit User Manual v1.0
Copyright © 2009 Nurve Networks LLC

Author
Joshua Hintze
Andre' LaMothe

Editor/Technical Reviewer
The "Collective"

Printing
0001

ISBN
Pending

Trademarks
All terms mentioned in this user manual that are known to be trademarks or service marks have been appropriately capitalized. Nurve Networks LLC cannot attest to the accuracy of this information. Use of a term in this user manual should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer
Every effort has been made to make this user manual as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an ***"as is"*** basis. The authors and the publisher shall have neither liability nor any responsibility to any person or entity with respect to any loss or damages arising from the information contained in this user manual.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

eBook License

Licensing, Terms & Conditions

NURVE NETWORKS LLC, . END-USER LICENSE AGREEMENT FOR XGS PIC 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 PIC 16-Bit Game Console Revision A. or greater.

Microchip MPLAB 8.15a 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**

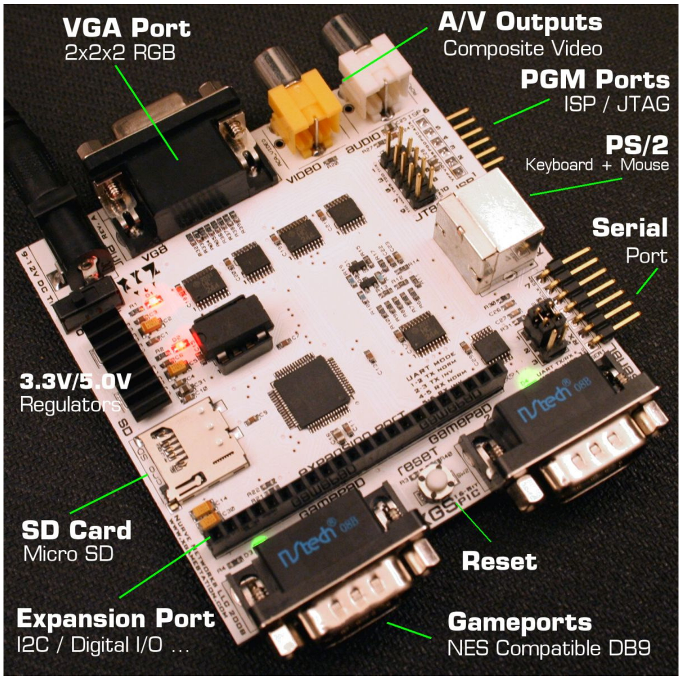
Inside the XGS PIC 16-Bit

User Manual and Programming Guide (SAMPLE)

Part I – Hardware Manual

1.0 XGS PIC Overview (SAMPLE)

Figure 1.1 (a) – The XGS PIC 16-Bit.



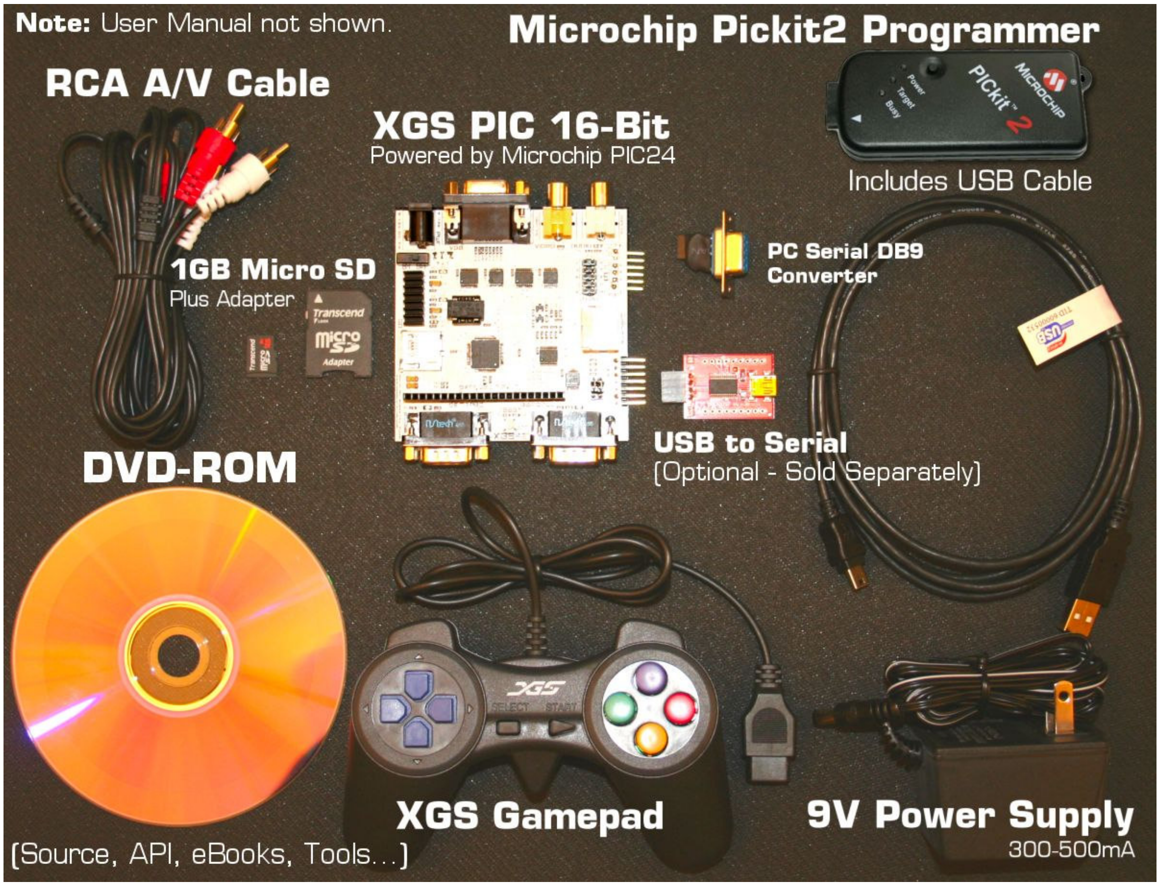
The **XGamestation PIC 16-Bit** (or simply put XGS PIC) is developed around the Microchip PIC24HJ256GP206 16-bit processor. Figure 1.1(a) shows an image of the XGS PIC with all the various functional units labeled. The XGS PIC game console has the following hardware features:

- 64-Pin TQFP Package of the PIC; runs 40MHz, however we will be over clocking.
- RCA Video and Audio Out Ports.
- HD15 Standard VGA Out Port.
- PS/2 Mouse and Keyboard Support.
- Two DB9 connectors that work with the XGS exclusive labeled gamepads, similar to NES gamepads.
- MicroSD card slot for large external non-volatile memory requirements.
- Single 9V DC power in with regulated output of 5.0V @ 500mA and 3.3V @ 500 mA onboard to support external peripherals and components (**Note:** the PIC is a 3.3V device with 5 Volt tolerant inputs).
- Removable passive XTAL to support experimenting with various reference clocks.
- JTAG debugging port (**Note:** the PIC24s do not support JTAG programming at this time, however they have the pins labeled for future support).
- In-Circuit-Debugging (ICD) programmers port. This allows the use of a cheap programmer to flash the chip and **debug!**
- 22 pin Expansion Port that exposes I/O, power, clocking, analog, PWM, interrupt, and a lot of GPIO pins.
- Serial I/O including RS-232, I2C/TWI and SPI communications.

The XGS PIC is essentially a fully operational computer given the microSD card as a removable storage device. By using very small surface mount devices, we have been able to pack a ton of power into the same space required by the precursor XGS pico edition, but with 100X more power. Next, let's inventory your Coding Kit.

1.1 Package Contents

Figure 1.1 (b) – XGS PIC kit contents.

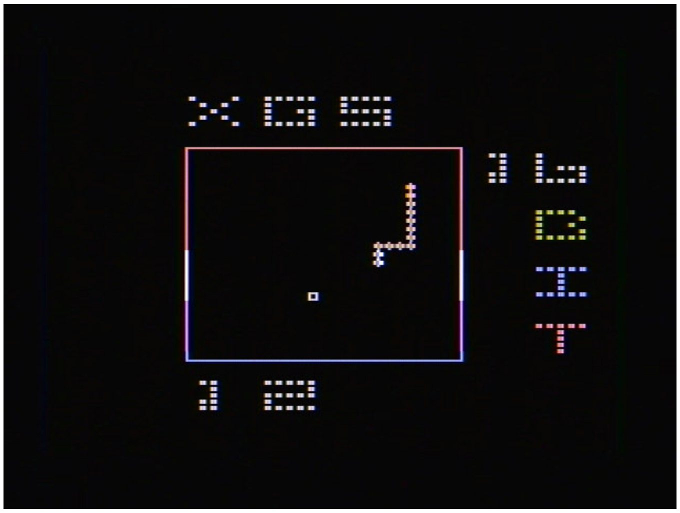


You should have the following items in your coding kit, referring to Figure 1.1(b).

- XGS PIC 16-Bit Game Console.
- DB9 RS-232 serial interface connector.
- 9V 500 mA, DC unregulated wall adapter with 2.1 plug and tip (+) positive, ring (-) negative.
- Microchip PICkit™ 2 programmer with USB cable.
- 1 GB blank micro SD card and standard size SD adapter.
- 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.
- User Manual (not shown).

1.2 XGS PIC "Quick Start" Demo

Figure 1.2 – The demo running.



The XGS PIC 16-bit is pre-loaded with a simple game demo programmed into the PIC24's flash, a screen shot is shown in Figure 1.2. 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 PIC on a flat surface, no carpet! Static electricity!

Step 2: Make sure the power switch at the back near the power connector is in the **OFF** position, this is to the **RIGHT**.

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

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

Step 7: Plug the XGS game controller into the **LEFT** controller port on the XGS.

Step 8: Turn the power on by sliding the ON/OFF switch to the **LEFT**.

You should see something like that shown in Figure 1.2. The actual program that is loaded into the PIC is located on your CD here:

DVD_ROM:\ XGSPIC \ SOURCE \ XGS_PIC_NTSC_SNAKE_V010.c

The demo is nothing more than a simple snake game that uses one of the bitmap NTSC engines for graphics, a simple sound engine, and the gamepad driver for reading in the user controls. The controls are shown in Table 1.1.

Table 1.1 – Tile demo game pad controls.

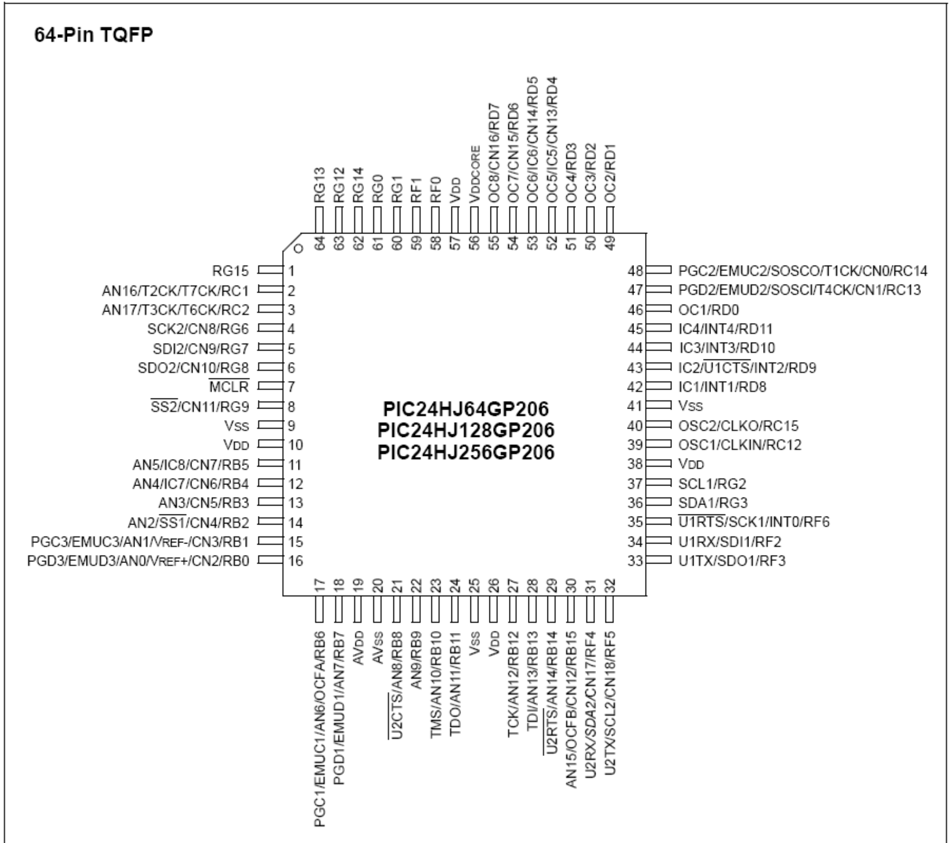
Action	Control
D-pad Left	Move Left
D-pad Right	Move Right
D-pad Up	Move Up
D-pad Down	Move Down

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.

1.3 The PIC24 16-bit Chip

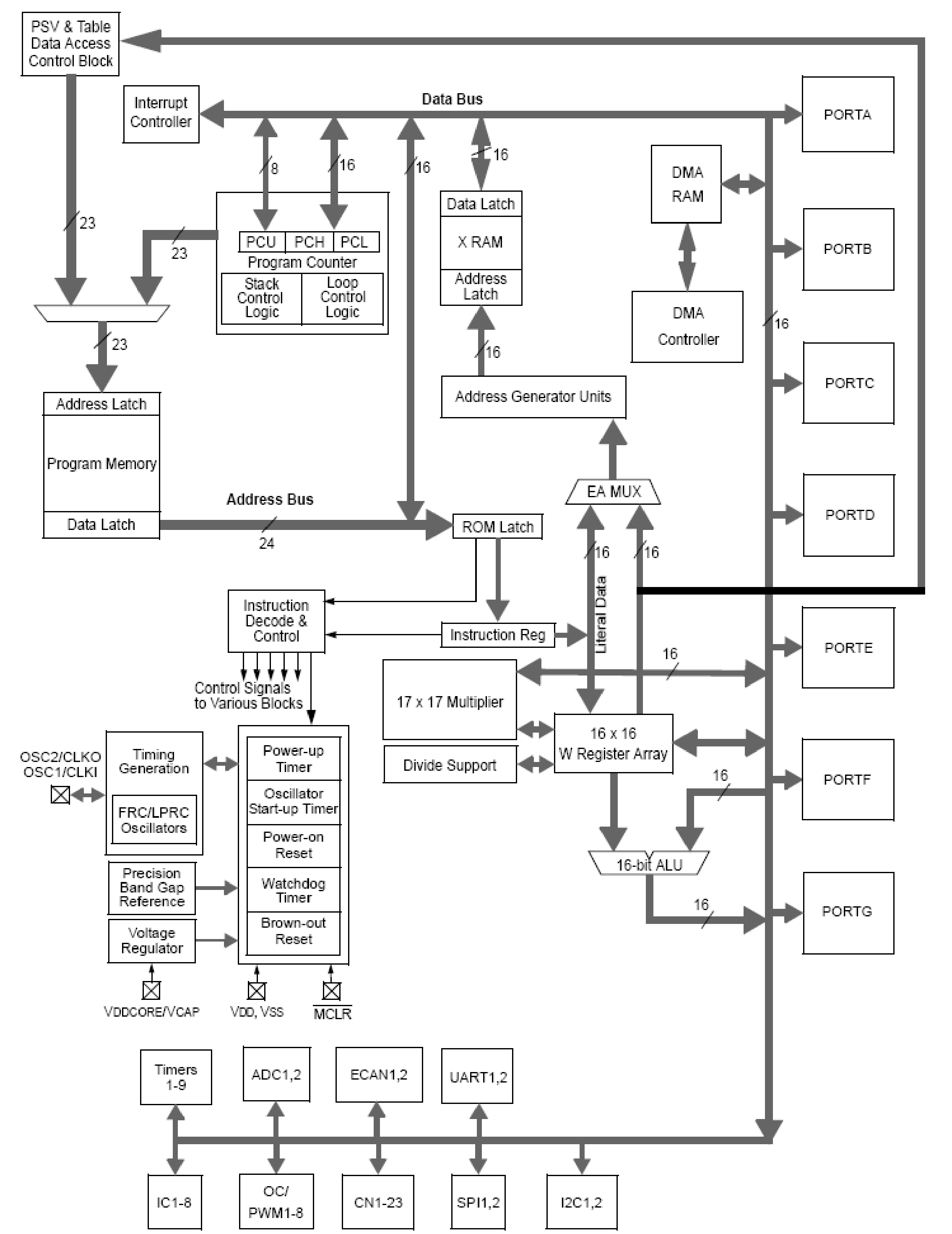
The microcontroller used in the XGS PIC game console is a 16-bit PIC24HJ256GP206. That's quite a long number! The reason for such a long part number is that Microchip makes hundreds of variants. Some with more ram, some with more flash, some with more peripherals. Our design methodology for this version of the XGS was to choose a 16-bit processor that had the most available flash and ram. There are a few pricier chips that have extra peripherals, like an enhanced CAN controller. Since you more than likely won't be hooking this up into your automobile for listening in on all the network traffic between your car sensors (CAN bus) we landed on the PIC24HJ256GP206. The PIC24 comes in a 64 and 100 pin **Thin Quad Flat Pack (TQFP)** surface mount package. 64 pins turned out to be plenty of pins for interfacing to all our different peripherals. There were even a few pins left over that are not connected. Figure 1.3 shows the packaging and pinout of the chip. Notice that some pins have dual, triple, or even quadruple pin names. This means that the pins can be multiplexed, or swapped, for different purposes. Another interesting thing to note is that power and ground (except for one side) comes in on all of the different sides of the TQFP. You'll actually find this common on chip packages like this. Figure 1.4 shows the PIC24's architecture in block diagram form. Don't stress about this block diagram too much. It basically gives you an overview of all that's available. A couple things you might have noticed in the diagram are the 17x17 single instruction multiply and divide hardware, ***very*** nice. Table 1.2 lists the pins and their function for the XGS PIC 16-bit game console.

Figure 1.3 – The PIC24HJ256GP206 packing for the 64-Pin TQFP.



Note: The PIC24HJ64GP206 device does not have the SCL2 and SDA2 pins.

Figure 1.4 – The PIC24 architecture block diagram.



Note: Not all pins or features are implemented on all device pinout configurations. See pinout diagrams for the specific pins and features present on each device.

Table 1.2 – The XGS PIC Pin Descriptions.

Pin Name	Pin Number	Description
I/O's located on the chip		
RB0	16	Video Color Sel0, VGA(R0).
RB1	15	Video Color Sel1, VGA(R1).
RB2	14	Video Color Sel2, VGA(G0).
RB3	13	Video Color Sel3, VGA(G1).
RB4	12	Video Intensity Sel0, VGA(B0).
RB5	11	Video Intensity Sel1, VGA(B1).
RB6	17	Video Intensity Sel2, VGA(HSYNC).
RB7	18	Video Intensity Sel3, VGA(VSYNC).
RF0	58	Video Saturation Sel0
RF1	59	Video Saturation Sel1
RD7	55	Audio Mono
RD10	44	Keyboard Clock
RD11	45	Keyboard Data
RG6	4	SPI Clock
RG7	5	SPI Master In/Slave Out **(MISO)**
RG8	6	SPI Master Out/Slave In **(MOSI)**
RG9	8	SPI CSn
RF2	34	Serial RX/ Secondary SPI (MISO)
RF3	33	Serial TX/ Secondary SPI (MOSI)
RF5	32	Secondary SPI CSn
RF6	35	Secondary SPI Clock
RD0	46	Gamepad 0 Data
RD1	49	Gamepad 1 Data
RD2	50	Gamepad SH/nLD
RD3	51	Gamepad Clock
RB10	23	JTAG TMS
RB11	24	JTAG TDO
RB12	27	JTAG TCK
RB13	28	JTAG TDI
RC13	40	In-circuit Serial Programmer **(ISP)** PGD
RC14	40	In-circuit Serial Programmer **(ISP)** PGC
Power and Reset		
Vss	9, 25,41	Ground.
Vdd	10, 26,38,57	Main power 3.3V.
AVdd	19	Analog Vdd.
AVss	20	Analog ground.
RESn	11	System reset active LOW.

VddCore	56	Internal regulated voltage output for 2.5V core.
Clocking		
CLKIN	39	Oscillator clock in
Expansion Port Interface		
RB14	1	Analog and General purpose I/O, source or sink 4mA.
RB15	2	Analog and General purpose I/O, source or sink 4mA.
RD8	3	Interruptible and General purpose I/O, source or sink 4mA.
RD9	4	Interruptible and General purpose I/O, source or sink 4mA.
RD5	5	PWM and General purpose I/O, source or sink 4mA.
RD4	6	PWM and General purpose I/O, source or sink 4mA.
RG3	7	I2C SDA w/ 10K pull-ups and General purpose I/O, source or sink 4mA.
RG2	8	I2C SCL w/ 10K pull-ups and General purpose I/O, source or sink 4mA.
33VCC	9	Main Power
5VCC	10	5V Power
GND	11	Ground
Main Clk	12	Oscillator clock output.
/Reset	13	System reset active LOW.
GND	14	Ground
RG6	15	SPI and General purpose I/O, source or sink 4mA.
RG7	16	SPI and General purpose I/O, source or sink 4mA.
RG8	17	SPI and General purpose I/O, source or sink 4mA.
RG9	18	SPI and General purpose I/O, source or sink 4mA.
RG12	19	General purpose I/O, source or sink 4mA.
RG13	20	General purpose I/O, source or sink 4mA.
RG14	21	General purpose I/O, source or sink 4mA.
RG15	22	General purpose I/O, source or sink 4mA.
Serial Interface		
N/A	1	No-connect
RX	2	Serial RX (normal/inverted)
TX	3	Serial TX (normal/inverted)
GND	4	Ground
5VCC	5	5 Volt power
SSCLK	6	Secondary SPI Clock
SCSn	7	Secondary SPI CS

In the remainder of this section I’m going to briefly describe the basic memory structure along with the startup and reset sequence. For the gory details I point to the verbose but almost robotic datasheet. Then for verbose++, I refer you to the reference manual that takes each section, be it SPI, Serial, PLL clocking, and has an entire chapter dedicated to it. The data sheet can be found on the CD at location:

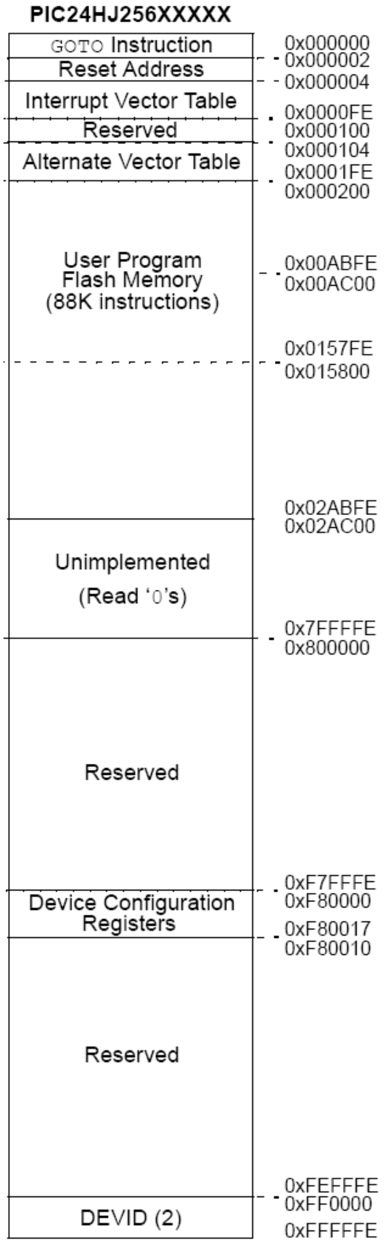
DVD_ROM:\ XGSPIC \ Documents \ PIC24HJXXXXGPX06_08_10.pdf.

The reference sections have their own directory located:

**DVD_ROM:\ XGSPIC \ Documents \ PIC24RefManual **

The PIC24HJ256GP206 is a 16-bit modified Harvard architecture with a *variable* instruction size, typically 1-2 words (1 word is 24-bits). The on chip SRAM memory is ***16K BYTES*** and is entirely accessible using the 16-bit addressing space. This is much nicer than having to play games with strange memory page registers (i.e. SX chips). The non-volatile memory consists of **256K BYTES** of flash program memory. This leaves us lots of storage space for game assets without having to worry about compression with run length encoding (RLE) or other schemes. In the datasheets, the flash is referred to as the ***program*** memory. Figure 1.5 shows the program memory map for all variants of the PIC24HJ256XXXXX chips. SRAM on the other hand is known as the ***data*** memory, with a layout given in Figure 1.6.

Figure 1.5 – Program Memory Map for the PIC24.



The program memory is divided into regions as you can see, most of them are pretty self explanatory. The first 512 locations provide reset and interrupt vectors. There is no moving these around since they are hardwired in. The next section is the User Program Memory that provides 88K of single word instructions for your usage. Since the program counter is 24-bits wide, we have a lot of extra space that is mostly filled with 0's or reserved sections that map to nothing. Possibly as more variants of the PIC24s come out they could be filled with more space for user programs.

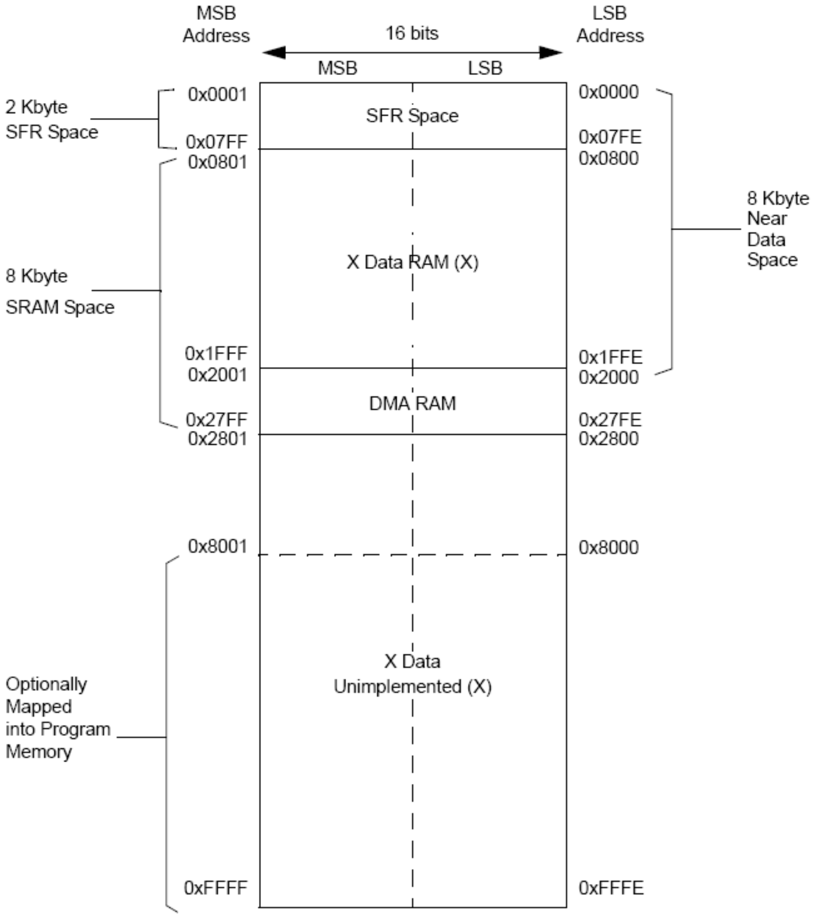
TIP	***Harvard*** as opposed to ***Von Neumann*** architecture are the two primary computer memory organizations used in modern processors. Harvard was created at Harvard University, thus the moniker, and likewise Von Neuman was designed by mathematician John Von Neumann. Von Neumann differs from Harvard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

The device configuration registers are located lower in memory space. These are essentially fuse bits for those who are familiar with AVRs or SX chips. They configure a number of settings for the device like: watchdog timer enable, debug port number, JTAG enable (even though this chip doesn't support it yet!), oscillator type selection, etc. These configuration registers are setup as #defines in your C code, more on this later.

Finally at the very bottom we have the device ID bits that specify what type of device this is. These are factory programmed and cannot be change (one would hope). The reason why we have device ID bits is because the core architecture for this chip is used in all PIC24 variants and this is the same core that runs in the dsPICs (minus some instructions for DSP processing).

NOTE	The Device ID bits have been problematic in the past. It turns out that in the latest silicon revision there is actually a BUG that can modify the device ID after it has already been factory programmed. The silicon errata states a work around for this, however some programming devices at the time of MPLAB v 8.14 and previous (the IDE for PIC processors) did not have this work around. So during prototyping phase we nuked a couple chips! Thankfully, Microchip has released MPLAB v8.15 with the work around implemented. Thus, try to use the latest version of MPLAB. The version on the CD is safe for your use.

Figure 1.6 – Data (SRAM) structure of PIC24's memory.



Our Data (SRAM) memory is segmented as illustrated in Figure 1.6. One thing that can be gleamed from the figure is that the PIC24 is using a **little endian** memory style. Just like the program memory, we don't have free reign of the entire memory space and certain sections of RAM will always map to special registers. The first section is the SFR region. The SFR acronym stands for the Special Function Registers. This includes things like I/O Port control and latch registers, SPI configuration registers, etc. Something worth mentioning is the concept of ***near*** and ***far*** data space. The near data space can only be contained in the first 8K bytes where as the far data region is anything beyond that. The purpose for the near and far space is to support instructions that don't quite have enough bits to reach the full extent. Thus, some instructions will not have the capability of working on data above the 8K memory address range. The work around would be to load the data into a local register and then make your computations and finally store it back. Obviously, this consumes more time and should be avoided in any critical timing code.

When we allocate variables, be it in assembly (ASM) or C code, we essentially assign them in near or far memory space. By default if you don't specify the location, the data is placed in near memory space. Now image you allocate a large portion of RAM for a video frame buffer of 200x224 pixels with 2 bits per pixel. This requires 200 * 224 * (0.25) = 11200 bytes of memory. If we just let the compiler place it into the default location (near space), we just ran out of near space, and the compiler will throw an error. Later on we will discuss how to place these variables into far memory space in both assembly and C code.

One final note on data memory. The PIC24 has 16K of data space. Each address points to a single byte. This requires an address range of 0x0000-0x4000. However we can use our 16-bit registers to access up to 0x0000-0xFFFF. So the developers at Microchip decided to use the upper 32K of address space to allow us to map it to the program (flash) memory. Here is where the term **modified** Harvard architecture comes into play; we can actually access and read our flash memory as if it was RAM. Very cool! There are some catches and a little additional setup that is required to ***page*** the required region into the RAM, but it is all clearly defined in the datasheets/manuals.

```
               ; Spend 14 clocks per pixel
PIXEL_LOOP:

               ; ----------------
               ; Pixels 0, 1, 2, 3
               ; ----------------
               ; Rendering Loop
               mov.b [W0], W2                      ; (1) Get the next 4 pixels (2-bits each)
               lsr W2, #6, W1                      ; (1) Shift right and store in W1
               and #0x03, W1                       ; (1) Mask off bottom two pixels
               mov.b [W1+W3], W2                   ; (2) Use indirect addressing to get assigned
color
               mov W2, PORTB                       ; (1) Copy to port B
               repeat #6                           ; (1) delay
               nop                                 ; (1+n)

               mov.b [W0], W2                      ; (1) Get the next 4 pixels (2-bits each)
               lsr W2, #4, W1                      ; (1) Shift right and store in W1
               and #0x03, W1                       ; (1) Mask off bottom two pixels
               mov.b [W1+W3], W2                   ; (2) Use indirect addressing to get assigned
color
               mov W2, PORTB                       ; (1) Copy to port B
               repeat #6                           ; (1) delay
               nop                                 ; (1+n)

               mov.b [W0], W2                      ; (1) Get the next 4 pixels (2-bits each)
               lsr W2, #2, W1                      ; (1) Shift right and store in W1
               and #0x03, W1                       ; (1) Mask off bottom two pixels
               mov.b [W1+W3], W2                   ; (2) Use indirect addressing to get assigned
color
               mov W2, PORTB                       ; (1) Copy to port B
               repeat #6                           ; (1) delay
               nop                                 ; (1+n)

               mov.b [W0++], W1                    ; (1) Get the next 4 pixels (2-bits each)
               nop                                 ; (1) delay
               and #0x03, W1                       ; (1) Mask off bottom two pixels
               mov.b [W1+W3], W2                   ; (2) Use indirect addressing to get assigned
color
               mov W2, PORTB                       ; (1) Copy to port B
               repeat #6                           ; (1) delay
               nop                                 ; (1+n)
```

To whet your appetite for programming, I listed a little bit of PIC24 assembly code above. This is an excerpt from the NTSC 160x192 2 bit per pixel graphics driver. This code is a portion of the main rendering loop. Notice how it’s important to count out the exact cycles taken by each instruction so that we can have repeatable delays between plotting pixels. Without this we would have some pixels larger than others. Below is a snippet of C code that makes use of our graphics and peripheral libraries. Notice how much easier this will be for those who are beginners to embedded processors. Even if they do not know C very well it’s still quite simple to understand. More on all of this latter in the programmers section.

```
       // Initialize our uart driver
       UART_Init(115200);            // Start off in 115200:8:N:1 mode
       KEYBRD_Init();
       SND_Init();
       MECH_TempSensorInit();


       SND_PlayTone(1000);           // Play 1K Hz Tone

       // Check for keyboard key press “F5”
       If(KEYBRD_IsKeyPressed(SCAN_F5))
       {
           SND_PlayTone(500);        // Play 500 Hz Tone
       }
```

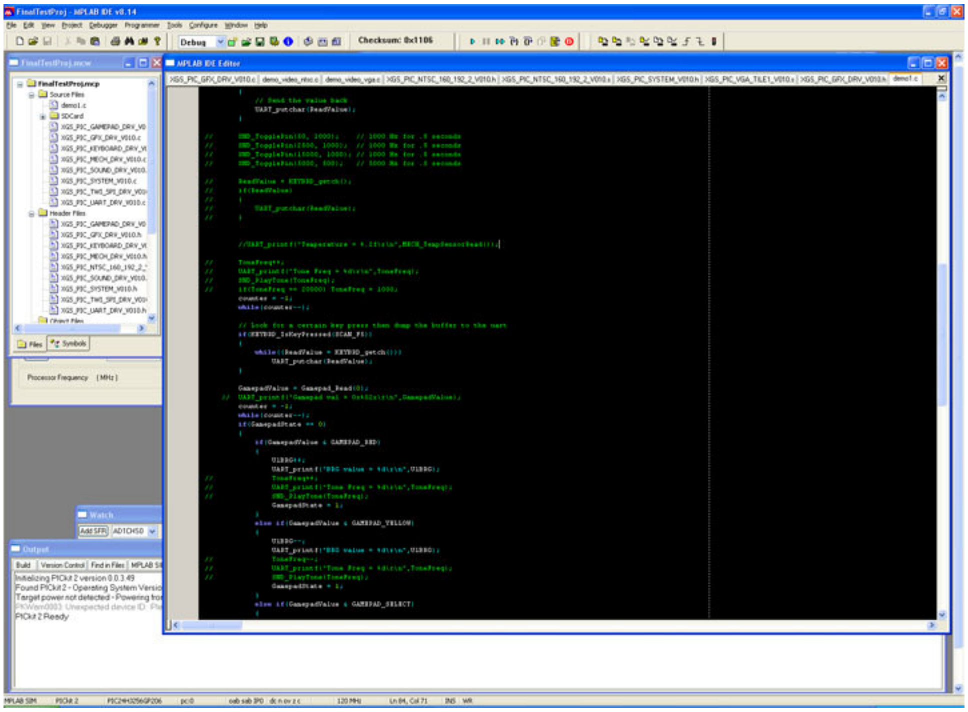
```
        // Put the temperature sensor value out to the UART terminal
        UART_printf("Temperature = %.2f\r\n",MECH_TempSensorRead());
```

The above snippet shows how simple the C code can be with a couple application programmer interface (**API**) libraries. Notice that initializing a UART is as simple as providing the baud rate. With the expansion port we can added mechatronics sensors like a temperature sensor or servo. Finally, we show an example of writing a string to the UART with the current temperature sensor value.

When writing code for the XGS PIC 16-bit game console you will be using Microchip's **I**ntegrated **D**evelopment **E**nvironment (**IDE)** called ***MPLAB***. A screenshot of MPLAB can be seen in Figure 1.7. MPLAB is a fully featured code editor with capabilities of programming, debugging, and simulating all of Microchip's PIC 8 and 16-bit line of processors. MPLAB IDE supports all the standard IDE features you come to expect from a professional software tool such as:

- Project files.
- Built-in debugger and programmer toolbars.
- Simulator with timing analysis.
- Trace capabilities for debugging output.
- Call stack monitoring.
- Debug variable watching.
- Simulated peripheral support.
- Editor syntax highlighting.

Figure 1.7 – The MPLAB IDE.



```
    {
        // Send the value back
        UART_putchar(ReadValue);
    }

//  SND_TogglePin(50, 1000);    // 1000 Hz for .5 seconds
//  SND_TogglePin(2500, 1000);  // 1000 Hz for .5 seconds
//  SND_TogglePin(15000, 1000); // 1000 Hz for .5 seconds
//  SND_TogglePin(5000, 500);   // 1000 Hz for .5 seconds

//  ReadValue = KEYBRD_getch();
//  if(ReadValue)
//  {
//      UART_putchar(ReadValue);
//  }


    //UART_printf("Temperature = %.2f\r\n",MECH_TempSensorRead());

//  ToneFreq++;
//  UART_printf("Tone Freq = %d\r\n",ToneFreq);
//  SND_PlayTone(ToneFreq);
//  if(ToneFreq == 20000) ToneFreq = 1000;
    counter = -1;
    while(counter--);

    // Look for a certain key press then dump the buffer to the uart
    if(KEYBRD_IsKeyPressed(SCAN_F5))
    {
        while((ReadValue = KEYBRD_getch()))
            UART_putchar(ReadValue);
    }

    GamepadValue = Gamepad_Read(0);
//  UART_printf("Gamepad val = 0x%02x\r\n",GamepadValue);
    counter = -1;
    while(counter--);
    if(GamepadState == 0)
    {
        if(GamepadValue & GAMEPAD_RED)
        {
            U1BRG++;
            UART_printf("BRG value = %d\r\n",U1BRG);
//          ToneFreq++;
//          UART_printf("Tone Freq = %d\r\n",ToneFreq);
//          SND_PlayTone(ToneFreq);
            GamepadState = 1;
        }
        else if(GamepadValue & GAMEPAD_YELLOW)
        {
            U1BRG--;
            UART_printf("BRG value = %d\r\n",U1BRG);
//          ToneFreq--;
//          UART_printf("Tone Freq = %d\r\n",ToneFreq);
//          SND_PlayTone(ToneFreq);
            GamepadState = 1;
        }
        else if(GamepadValue & GAMEPAD_SELECT)
        {
```

```
Initializing PICkit 2 version 0.0.3.49
Found PICkit 2 - Operating System Versio
Target power not detected - Powering fro
PKWarn0003: Unexpected device ID: Ple
PICkit 2 Ready
```

Since PIC processors have become very popular globally, there are also many different third party tools, compilers, and programmers. For the sake of keeping all of the software tools consistent we are going to stick with the Microchip line of products. In that regard we will also be using a C compiler and Assembler provided by Microchip that is specifically made for the PIC24 processors. The PIC24 assembly language tool chain is called ASM30. The C compiler is called C30 but with recent changes to the naming conventions by Microchip we have a new name – PIC24 C compiler. Regardless of the name, the fact remains that we will be using a command line compiler and assembler that integrates into the MPLAB IDE environment. This all becomes very seamless and we will cover the installation procedures in the first section of the Programming Manual Section.

NOTE	The C compiler of for the PIC24 processors is actually a modified version of the GNU GCC compiler. There are also some external libraries and header files created specifically from Microchip for the PICs. What this means is that Microchip charges money for the C compiler. However, Microchip also provides a **"free"** student version. The catch with the student version is that the first 60 days it is fully featured. Afterwards the C compiler disables the higher optimization levels and leaves us with only level 0 (no optimization) and level 1 (slight optimization).

The IDE contains projects that are composed of Assembly files, C files, object files, library files, and linker scripts. The working method is to write all of your code in MPLAB, and then compile and assemble it using the ASM30 and C30 compiler we setup and then use a device programmer to erase, program, and verify the flash memory on the PIC24. MPLAB allows in-circuit debugging capabilities as well as processor simulation. Debugging the hardware PIC24 chip requires a debugging capable programmer. A inexpensive programmer with this ability is the PICKit 2™ In-circuit Serial Programmer (**ISP**) that we will discuss in Section 5.0. In addition to hardware debugging you also have access to an excellent simulator built into MPLAB. The simulator works much faster as far as stepping through and watching variables as opposed to the debugger. There is even a stop watch timer that allows users to time critical sections of code for analysis. This is beneficial when writing video driver code and allows you to **double** check your instruction cycle counts. Of course don't always trust the simulator, trust yourself then check it with the simulator. The downside with the simulator is that you will not be able to test peripherals like the UARTs, gamepads, keyboards, etc.

In the future somebody, *or maybe you*, will write an interpreter that can take input from either the serial port or maybe an onscreen editor that provides a simple basic language. The software could even use the SD card to store saved program files for loading and sharing user created content over the internet. There are endless possibilities but right now let's discuss how the system starts up.

1.3.1 System Startup and Reset Details

On startup/reset the program counter (**PC**) is set to 0x0000. Referring back to Figure 1.5 of our program memory, our linker always places a **GOTO** instruction at address 0x0000 that **jumps** execution to the C startup routine that performs operations such as: setting the default values for all our initialized data, clearing out the zeroed memory (.bss) sections, and calling our main function. The initial configuration of the chip is defined by the ***device configuration registers*** which are controlled by the programming of the chip. In our case, the registers are mostly default, no watchdog timer, clock switching enabled, secure memory segments off, etc.

Once our application **main()** is called we do a couple housekeeping chores. The first step is to configure the clock source. Initially the PIC24 is clocked from an internal **F**ast **R**esistor **C**apacitor (FRC) clock at 7.32 MHz. Since our chip can operate up to 40 MHz and we'll even be pushing above that in most cases, we need to switch the clock settings. To do this we'll setup registers like the Phase Lock Loop (PLL) and clock configuration and then initiate a clock switch. For the end user this is handled in the XGS PIC system library we provide. After the clock is configured the next steps are application dependent. If the user requires video, the video is initialized. Likewise, if the application requires a serial UART, the onboard hardware UART is configured. More details on the application specific startup details in the programming manual sections.

End Sample

15.0 Graphics Programming (SAMPLE)

In this section we will explore the “**Graphics Driver**” and all of its supporting files that are included in the XGS PIC API. The graphics code sections are comprised of mixed **C** and **Assembly** interspersed throughout many different source files. All video is created through “software generation” since the XGS PIC does not have a dedicated GPU. Having no GPU was part of the design decision in order to *show off* the raw processing power of the PIC microcontroller.

Even though a number of graphic’s drivers have been written, there is nothing stopping you from improving or adding your own different graphics modes with varying resolutions. Before you go off making your own versions of Doom, it would benefit you to see how we created the graphics drivers for the XGS PIC. For those of you who are completely new to video programming and graphics rendering we will cover some of the basics. Thus, the following topics will be covered in this section:

- 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.

Computer graphics in general is a huge subject and we won’t even begin to cover all the necessary fundamentals in this little manual. Many books have been written on the subject that consist of 1000+ pages of information goodness. So more or less we will cover our XGS API and some basic bitmap and tile graphics theory. When you combine this section with a number of demos described near the end of this manual you can probably piece together a lot of information and understand what is going on. However, if you really want to dive into the world of computer graphics make sure to check out the following **free** ebooks on the DVD:

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

The books focus on DOS and Windows (DirectX) video game programming and even though the microcontroller isn’t running an operating system, much can still be learned from the ebooks. The ebooks are located here:

DVD_ROM:\ XGSPIC \ DOCS \ eBooks

15.1 Graphics Drivers and System Level Architecture

Before we begin, Table 15.1 lists the graphics library source files once again for reference, so you can see how much there is to talk about.

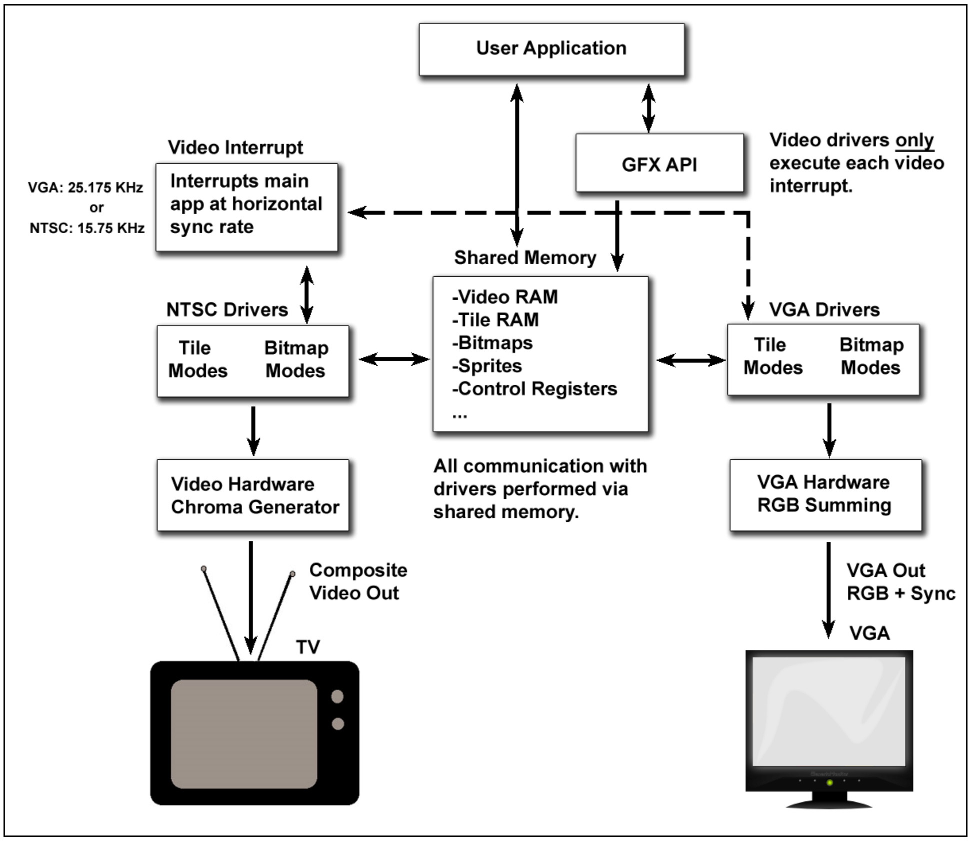
Table 15.1 – Graphics library and driver files.

Library Module Name	Description
High Level Graphics Modules	
XGS_PIC_GFX_DRV_V010.c|h	Graphics initialization, setup, and bitmap rendering primarily.
NTSC Bitmap and Tile Engines	
Bitmap Engines ►	
XGS_PIC_NTSC_160_96_4_V010.s|h	160x96 4-bits per pixel (16 color) NTSC graphics driver.
XGS_PIC_NTSC_160_192_2_V010.s|h	169x192 2-bits per pixel (4 color) NTSC graphics

	driver.
Tile Engines ►	
XGS_PIC_NTSC_TILE1_V010.s|h*	160x192, 20x24, 8x8 tiles resolution, **6** sprites, NTSC tile driver.
XGS_PIC_NTSC_TILE2_V010.s|h*	168x200, 28x25, **6x8** tiles resolution, **4** sprites, NTSC tile driver.
VGA Bitmap and Tile Engines	
Bitmap Engines ►	
XGS_PIC_VGA_128_120_2_V010.s|h	128x120 2-bits per pixel (4 color) VGA graphics driver.
XGS_PIC_VGA_128_120_4_V010.s|h	128x120 4-bits per pixel (16 color) VGA graphics driver.
XGS_PIC_VGA_160_160_4_V010.s|h	160x160 4-bits per pixel (16 color) VGA graphics driver.
XGS_PIC_VGA_200_200_2_V010.s|h*	200x200 2-bits per pixel (4 color) VGA graphics driver.
Tile Engines ►	
XGS_PIC_VGA_TILE1_V010.s|h*	176x240, 22x30, 8x8 tiles resolution, **3** sprites, VGA tile driver.
***Denotes modes that do not have multiple palettes.**	

Take a look at Figure 15.1 below that shows a system level architecture between the graphics drivers and the user application.

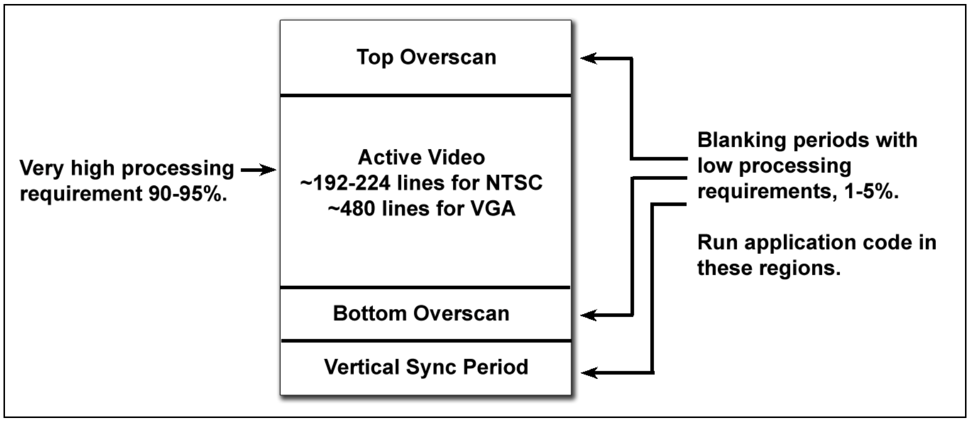
Figure 15.1 – Graphics library architecture and relationship to applications.



As you can see by the figure above, there are a few different areas we need to cover in order to understand the graphics driver. The first thing to understand is that the video is rendered completely in software using hardware timer interrupts. For VGA a timer on the PIC processor is set to run at 25.175 KHz that correlates to the horizontal line rate of VGA video. For NTSC this timer runs at 15.75 KHz. A user application would include and compile in a graphics mode (VGA or NTSC and Bitmap or Tiled) along with the standard graphics library "**XGS_PIC_GFX_DRV_V010.c|h**". During setup of the application, the user would initialize the graphics library and start the timer interrupt. Controlling the actual rendered image is handled through a combination of shared video memory and function calls between all 3 pieces: user code, the standard graphics library, and graphics mode assembly driver.

The graphics mode assembly driver, when compiled into the application, replaced the PIC24's Timer 1 **I**nterrupt **S**ervice **R**outine (**ISR**). All that this means is that when Timer 1 expires a function in the assembly driver is called. It then becomes the responsibility of the assembly routine to draw out a single line of video whether it is a blanking line, active video line, or a vertical sync. This is illustrated in Figure 15.2.

Figure 15.2 – Application code runs during blanking periods for the most part.


Top Overscan
Very high processing requirement 90-95%.
Active Video
~192-224 lines for NTSC
~480 lines for VGA
Bottom Overscan
Vertical Sync Period
Blanking periods with low processing requirements, 1-5%.
Run application code in these regions.

Since the assembly driver is responsible for drawing video it more or less needs to take up 100% of the CPU time to ensure that the video does not stop. You would think that there would be no left over time for the user application code. However, during the blanking regions and the vertical retrace, the processor is given over to user application code for a period of time. Fortunately for us, the processor runs at 40+ **M**illions of **I**nstructions **P**er **S**econd (**MIPS**) yielding a fair amount of time to update the screen and do other work in between frames of video.

In order for the user code to know when the graphics library is in a blanking region, the graphics driver exports out a number of global variables, one of which is a flag that is set during the blanking region and cleared during the rendering/active region. Then using some macros defined in the graphics header files the user application blocks the main C processing loop until the video has entered into the blanking region.

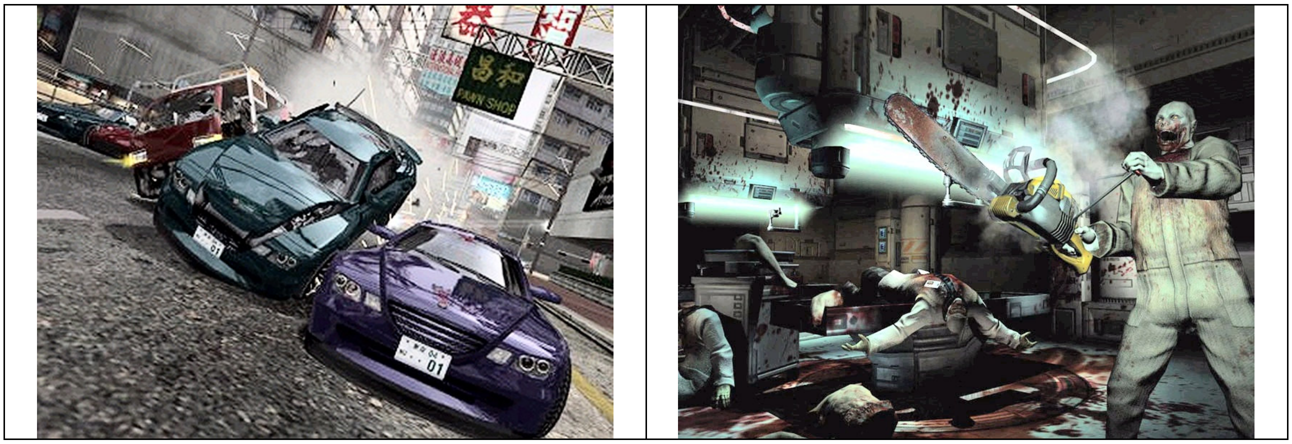
The reason why we want to stall the main execution loop during video rendering is because of interrupt **jitter** and shared memory accessing. If the user application was allowed to run at will, the interrupt routine, when activated, would stop the C application where ever it is currently operating. Normally this is not a big deal and it is by definition what an interrupt is all about (i.e. it interrupts normal code operation.) However, the PIC processor has instructions that may take 1, 2, or even more clock cycles. If an interrupt occurs during a multi-cycle operation it first waits until the instruction completes before the interrupt is serviced. This means that the assembly driver might fall one or more instructions behind and that single clock cycle is enough to cause video jitter. The other more obvious reason why we don't want to operate during the video interrupt is that accessing and changing the video ram while the PIC is also drawing out of the video ram would cause the screen to be partially updated or changed while the other portion of the screen remains old.

Now, take a look again at Table 15.1 and notice that there are a lot of different graphics modes. The reason for this is that graphics modes (bitmap in particular) require an enormous amount of video ram. With the PIC24 microcontroller we are limited to 16K Bytes of RAM (this is actually quite a bit for a microcontroller) in which to fit our user application and video memory data. Also note that there are different types of drivers including VGA and NTSC (television) and among those there are bitmap and tile mode drivers. The NTSC drivers are what you will primarily use when making video games since you can hook them up to your big screen TV's and show friends and family. The VGA drivers are meant to connect to any VGA compatible monitors and are good for simulation and very clean looking graphics. Because of all the different possibilities we have made a number of permutations of the graphics mode drivers such that you can pick and choose which best fits your system and memory constraints. Once you have mastered how the drivers work you may feel inclined to create your own versions for whatever your specific needs.

If the terms ***bitmap*** and ***tile*** mode graphics are a little foreign for you, do not fear. We are going to briefly talk about what they mean and then in the following sections we will discuss the technical concepts in length.

Bitmap graphics means that you have an image on a screen that you can individually address every pixel on the display. Figure 15.3 depicts a couple screen shots from bitmapped display games. As you can see, there is no regularity to the image, it's got a lot of information and color, every single pixel is generated each frame and is individually addressable.

Figure 15.3 – 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!

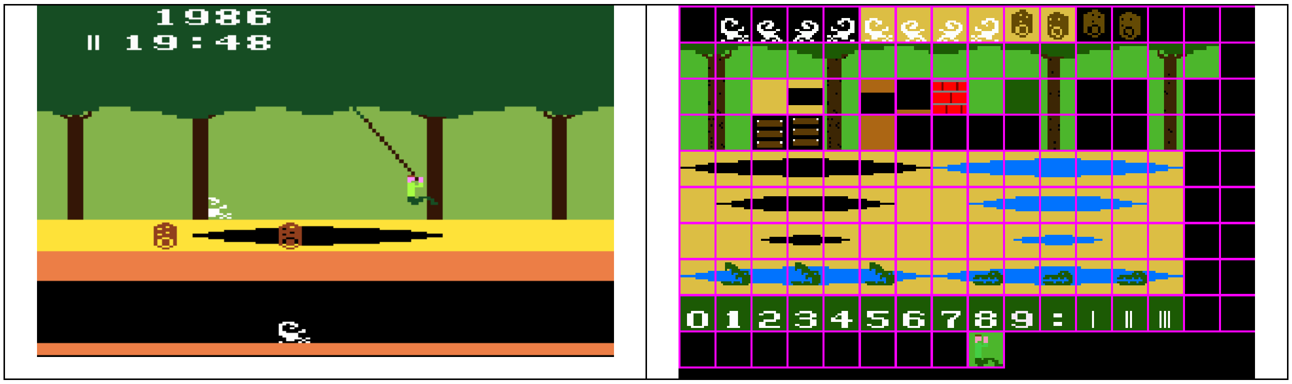
Which is quite a bit of memory even on a PC. Moreover, to facilitate animation you might need a ***"double buffered"*** display. In other words, while you are looking at one image, the computer generates the next one on an off screen buffer, then during the vertical blank or retrace of the video system, the images are flipped or copied. This doubles the amount of memory you need! Therefore, bitmapped graphics on embedded systems like the XGS PIC 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 160x192 with 2-bits per pixel or 4-pixels per byte, thus the memory requirements for this mode are:

160 * 192 / 4 = 7680 bytes!

Which is much better and fits in the 16K RAM of the PIC24. Of course, you might be saying, ***"160 x 192?, my cell phone has a larger screen!"*** And you would be correct, but you would be surprised how well 160x192 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 15.4(a) below shows a tile mapped image while Figure 15.4(b) shows the tiles that make up the image.

Figure 15.4 – The Atari 2600 classic "Pitfall" along with some of the tiles templated.

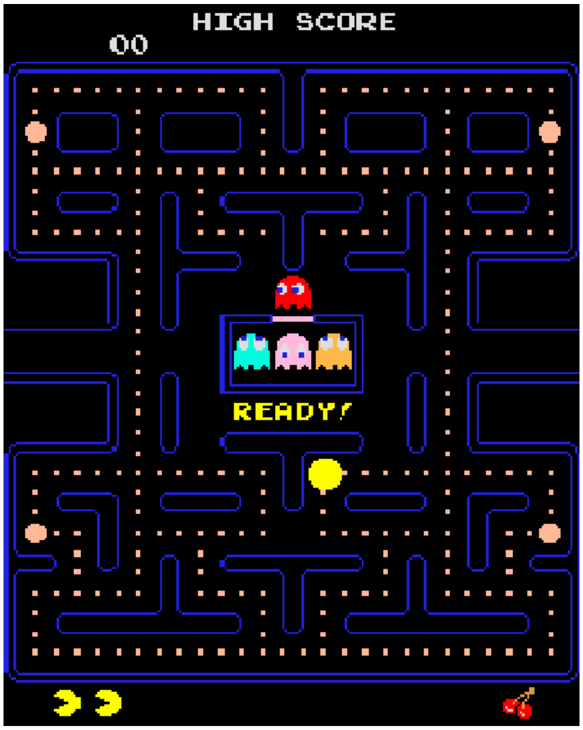


Referring to Figure 15.4, you can see how this system works. As an artist you might start by creating some sketches of the kinds of screens and levels you want then you might start drawing some mock ups of the imagery. Then at some point, you begin ***"ripping"*** the imagery into a set of common rectangular tiles usually 8x8 or 16x16 pixels in size and then template them into a ***"tile set"*** (Figure 15.4(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, it's very hard to move a character smoothly on the screen. Moreover, it's 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 15.5 shows a screen shot of Pacman for reference.

Figure 15.5 – A screen shot of the famous game Pacman showing a tiled background with sprite based characters.



The tile mapped drivers developed for the XGS PIC 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 PIC 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.

15.2 Bitmap Graphics Primer and Driver Overview

In this section we are going to focus on bitmap graphics and by doing so we will cover one of the graphics drivers in detail. Since all of the bitmap graphics drivers we have developed are similar, we can focus on one only and you should be able to understand all the other bitmap drivers, the same is true for tile engines. The important information to take away is how the bitmap driver operates including the initialization, relationships between memory and palette indirections, and finally how the available data structures work. Both NTSC and VGA graphics drivers are very similar, especially when viewing them from a high level language like “C”. The notable differences are in how their colors are encoded in the **V**ideo **RAM** (**VRAM**), and the different timing characteristics requiring a different clock setup to be initialized. Listed below are the current set of bitmap graphics drivers provided in the XGS PIC library:

▪ XGS_PIC_NTSC_160_96_4_V010.s|h	160x96 4-bits per pixel (16 color) NTSC gfx driver.
▪ XGS_PIC_NTSC_160_192_2_V010.s|h	160x192 2-bits per pixel (4 color) NTSC gfx driver.
▪ XGS_PIC_VGA_128_120_2_V010.s|h	128x120 2-bits per pixel (4 color) VGA gfx driver.
▪ XGS_PIC_VGA_128_120_4_V010.s|h	128x120 4-bits per pixel (16 color) VGA gfx driver.
▪ XGS_PIC_VGA_160_160_4_V010.s|h	160x160 4-bits per pixel (16 color) VGA gfx driver.
▪ XGS_PIC_VGA_200_200_2_V010.s|h	200x200 2-bits per pixel (4 color) VGA gfx driver.

NOTE

All of the bitmap drivers contain an 8x8 pixel map that allows a user to assign different color palettes to a region, **EXCEPT** the 200x200 VGA bitmap driver (XGS_PIC_VGA_200_200_2_V010.s|h). The reason for leaving out the palette map indirection, explained later, was that there was not enough clock cycles to left over to perform the necessary load operations.

We will now look at the 160x192 2-bit per pixel NTSC bitmap driver **XGS_PIC_NTSC_160_192_2_V010** as the example driver of our discussion.

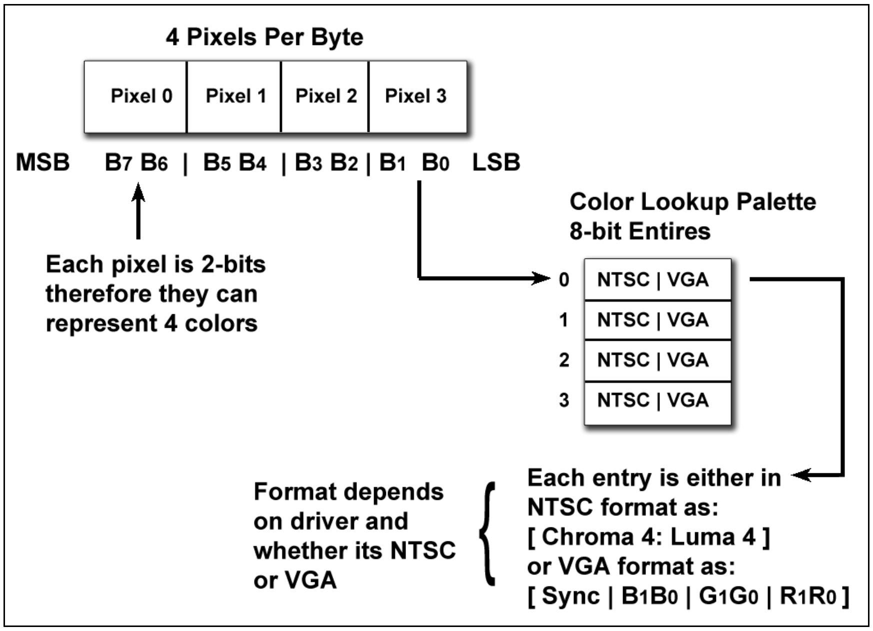
The first thing to realize is that each bitmap driver contains a different resolution, in our example our resolution is **160x192**. Also, this driver contains only 2 bits of memory per pixel. By using 2 bits for our color selection we are left with only 4 possibilities **(2^2 = 4)** and a single byte of VRAM will contain 4 pixels. The color encoding for a 2-bit per pixel is given in Table 15.2

Table 15.2 – 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 15.6.

Figure 15.6 – Each byte packs (4) 2-bit pixels MSB to LSB.

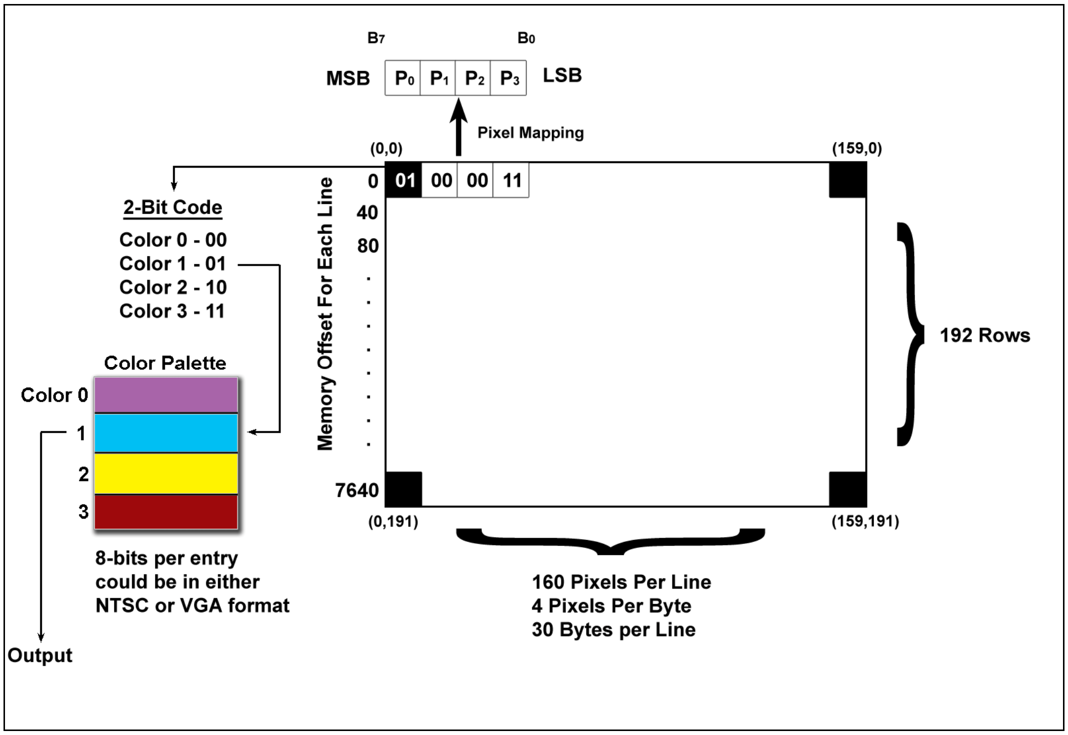


Notice how we labeled each entry in Table 15.2 as Color X with the available options 0 through 3 and we did not label them as specific colors like black, red, blue, etc. The reason for this is the concept of **indirection** that we have built into the graphics drivers. On startup and anytime afterwards, a user assigns a color palette to the bitmap driver. Then the user creates a bitmap display based on color **references** in the video ram and not actual colors. By using color indirection we gain a number of advantages. First color indirection makes it easy to port code between VGA and

NTSC bitmap graphics drivers, since it more or less requires the programmer to change only what is in the small color palettes opposed to the structure of the VRAM. Another bonus is that developers can create amazing effects using color palette animations. Imagine having a screen full of red and black bricks and you wish to change them from red bricks to blue bricks. Without color indirection the user would need to modify the entire VRAM display replacing every pixel that was once red with a blue pixel. Depending on the speed of the operation it might not actually be possible in a single video frame to do so. Now using color indirection the user would only need to swap the red color with a blue color in the color palette and then the entire screen is updated! We will see an example of this in the demo sections below.

Now a 4 color display may be OK in some applications and retro games of old, but we want the opportunity to have more colors. The answer to this problem is in palette maps. The idea is that we take the bitmap display and divide it up into 8x8 pixel groups. Then we assign each pixel group its own color palette. This allows us to have a much larger selection of colors for display on the screen without increasing the 2-bit per pixel requirement. All of these concepts might take a while to sink in so let's take a look at Figure 15.7 which shows how all of this ties together.

Figure 15.7 – Generalized architectural overview of the 160x192 NTSC bitmap mode.

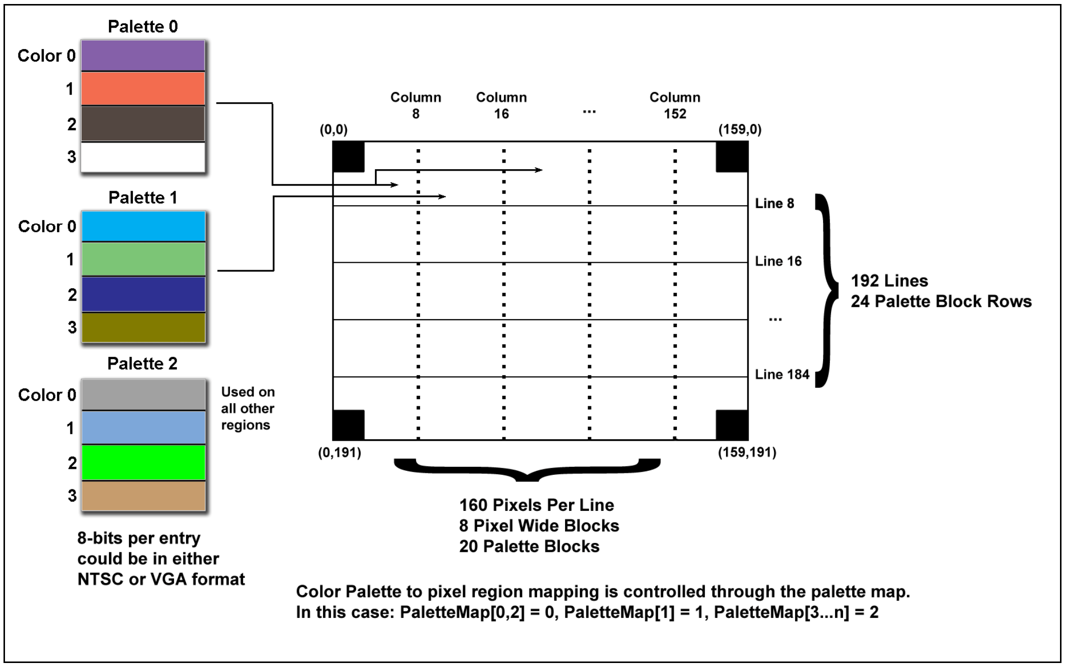


The first thing I want to point out is the large bitmap. In this driver we have a bitmap with of 160 and a height of 192. This leads to a coordinate system that ranges from (0…159 x 0…191). The top left hand corner contains the coordinate pixel (0,0) while the bottom right corner has the coordinate (159,191). If we wanted to modify the memory location of these pixels we would need to modify a portion of the byte contained at **VRAM[0]** for pixel 0 and **VRAM[(160*192)/4 – 1]** for the last pixel.

Contained within each byte are 4 pixels. Now we have a choice, we can either arrange the pixels 0 through 3 from **M**ost **S**ignificant **B**its (**MSB**) to **L**east **S**ignificant **B**its (**LSB**) or **LSB** to **MSB**. It is a little bit easier to draw out on graph paper the scenario where pixels are plotted from MSB to LSB. In the case of the XGS PIC library we have chosen to list them as pixels 0 to 3 from MSB to LSB as shown in Figure 15.7. **THIS IS OPPOSITE OF THE XGS AVR 8-bit!** The reason why we can get away with this on the XGS PIC is because it has a barrel shifter, making it so that we can perform multiple bit shifts in a single clock instruction. However, it would be very easy for someone to dig into the bitmap driver and reverse the order if needed.

Again referring to Figure 15.7, notice how Byte 0, Pixel 0 has added detail on the left side that shows pixel 0 containing the binary value (**01**). When the bitmap driver reads this memory entry it locates the value for Color 1 and draws that color as the ***final*** value to the screen. When the bitmap driver continues rendering pixels 1 and 2 it will use Color 0 and finally for pixel 3 it will use Color 3. These are all considered "**palette entries**" in the color table used for byte 0. This is very important to understand because it implies that the VRAM doesn't know about the final color used for rendering. Now on top of this byte level indirection we are going to add one more step and that is to subdivided regions of 8x8 pixels into a palette map as shown in Figure 15.8.

Figure 15.8 – Palette Map Encases Every 8x8 pixel block.


Palette 0
Color 0
1
2
3
Palette 1
Color 0
1
2
3
Palette 2
Color 0
1
2
3
Used on all other regions
8-bits per entry could be in either NTSC or VGA format
Column 8
Column 16
...
Column 152
(0,0)
(159,0)
Line 8
Line 16
...
Line 184
192 Lines
24 Palette Block Rows
(0,191)
(159,191)
160 Pixels Per Line
8 Pixel Wide Blocks
20 Palette Blocks
Color Palette to pixel region mapping is controlled through the palette map.
In this case: PaletteMap[0,2] = 0, PaletteMap[1] = 1, PaletteMap[3...n] = 2

In the figure above we see the bitmap divided up into 8x8 pixel blocks and in the case of our bitmap driver example, with resolution 160x192 pixels, we will have a palette map with size 20x24. Also in this example we are going to say that the user has a need for only 3 different color palettes. This first pixel region uses color palette 0. The second pixel region uses color palette 1. However, the third pixel region also uses color palette 0. Perhaps the user required the same colors as color palette 0 or a portion of them. Regardless of the reason, this illustrates that pixel regions can share color palettes thus reducing the total size of ram required to store the color palettes. All other palette map regions use color palette 2.

The palettes themselves contain the final color value that is pushed out to the graphics port. For NTSC this contains a 4-bit **LUMA** and a 4-bit **CHROMA** [LUMA:CHROMA]. With VGA the color entry is a 6-bit **RGB** value. The nice thing about this is that the color palette contains abstract ***"data"*** as far as the driver is concerned. Meaning that if color entry 0 is approximately the same red in a VGA and NTSC application, the video for that color will look the same. This leads to easily porting high level application code from a VGA monitor to an NTSC television. An example of a section of C code that sets up three (4-entry) color palettes is given below.

```
// This is just a simple example of filling up 3 different palettes.
// Notice that the palettes must be in continuous memory. The color
// #defines can be found in the graphic driver's header.

unsigned char g_Palettes[MIN_PALETTE_SIZE*3];


// Somewhere in the main() function...

// Palette 0
```

```
g_Palettes[0] = NTSC_BLACK;
g_Palettes[1] = NTSC_RED;
g_Palettes[2] = NTSC_YELLOW;
g_Palettes[3] = NTSC_GRAY7;

// Palette 1
g_Palettes[4] = NTSC_BLACK;
g_Palettes[5] = NTSC_ORANGE;
g_Palettes[6] = NTSC_BLUE;
g_Palettes[7] = NTSC_GREEN;

// Palette 2
g_Palettes[8] = NTSC_BLACK;
g_Palettes[9] = NTSC_WHITE;
g_Palettes[10] = NTSC_GRAY7;
g_Palettes[11] = NTSC_GRAY4;
```

If we were to write this code using a VGA driver the only change that would be required would be to replace the NTSC_COLOR #defines with their VGA_COLOR equivalents.

In addition to setting up the color palettes a user must also setup the palette map for the bitmap display. This is nothing more than a 16-bit short array that is sized appropriately for the graphics driver being used. The array must be an array of C ***shorts*** for the possibility that the user has more than 256 palettes. If we were to use Figure 15.8 as an example palette map, the following code would be required.

```
// Allocate the entire palette map array as a short array

unsigned short g_PaletteMap[PALETTE_MAP_SIZE];  // PALETTE_MAP_SIZE is defined for us in the
                                                     graphics header

// Somewhere in the main() function...

// Fill the entire palette map with 0 to start with
memset(g_PaletteMap, 0, sizeof(g_PaletteMap));

// Individually assign some different palette tables
g_PaletteMap[1] = 1;
g_PaletteMap[18] = 2;
g_PaletteMap[19] = 2;
```

Now obviously we are not adding all of these colors and palettes to our system without any penalty. Each palette and the entire palette map consume precious microcontroller RAM. You might be wondering whether the benefits of adding a palette mapped color system outweigh using 4-bits per pixel instead of 2-bits per pixel + palette maps. Let’s do some calculations. If we were to increase the 160x192 driver from 2-bits to 4-bits we are effectively doubling the current memory requirement. Right now the 160x192x2 driver requires:

160 * 192 = 30720 / 4 = 7680 bytes

Now doubling that gives us a requirement of **15360** bytes. If the user’s end application used only the 3 palettes we wrote code for above, the memory footprint for the color palettes would be:

3 palettes * 4 bytes per palette = 12 bytes

Plus the space for the palette map in 160x192 driver:

20 * 24 * 2 bytes per entry = 960 bytes

So our final total is **7680 + 12 + 960 = 8652**. This is significantly less than going to a 4-bit per pixel color value of the same resolution.

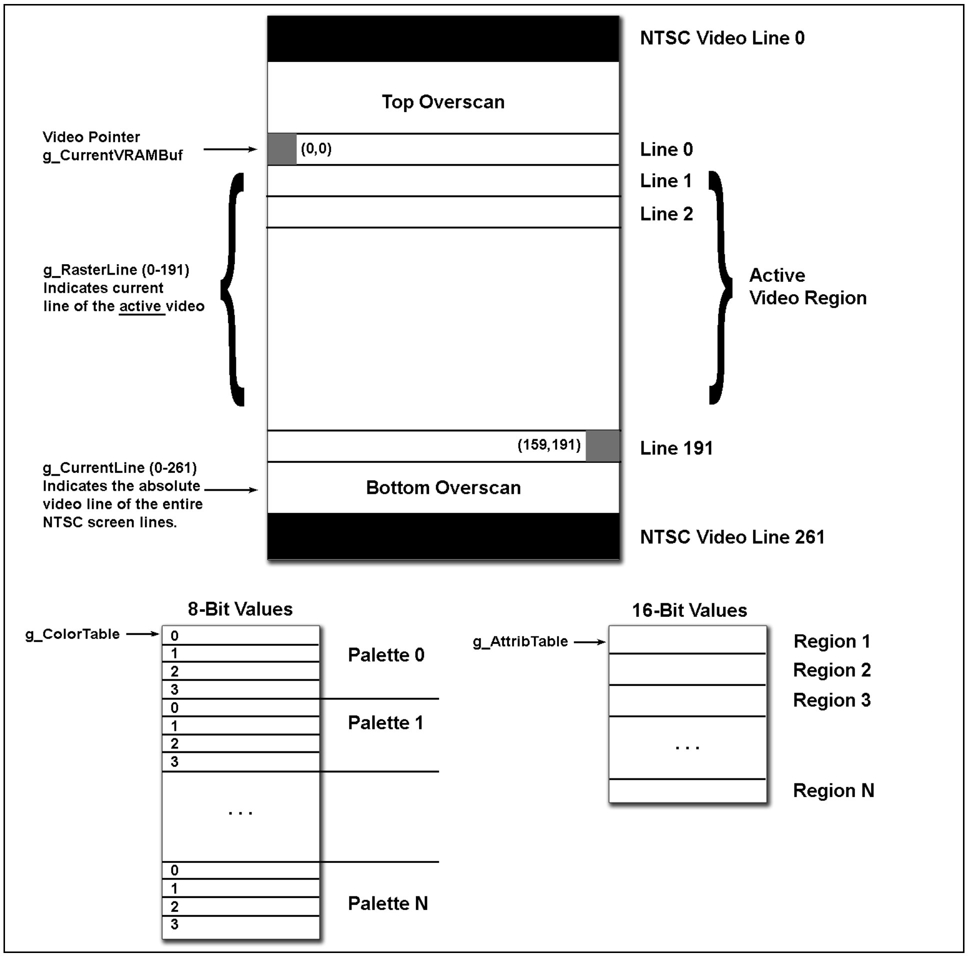
Let’s review now, before moving forward. There are both NTSC and VGA bitmap drivers. They are both roughly the same architecturally; they both use 2-bits or 4-bits per pixel which are color references, not the actual colors. The colors are always in a color table or palette. A palette map is always assigned to a bitmap driver (**except** for VGA

200x200x2 where it uses only a single palette) and it represents an 8x8 pixel region. 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 2 or 4 to a byte, you have to not only compute the proper address of a pixel, but the proper 2-bit or 4-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.

15.2.1 Accessing the Bitmap and Manipulating Pixel Data

Let's continue our discussions by deconstructing the 160x192 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 15.9 below, it illustrates the 160x192 graphics mode along with some data structure labels.

Figure 15.9 – The 160x192x2 (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_PIC_160_192_2_V010.s**. Here's an excerpt from the ASM file that lists the variables in question:

```
;//////////////////////////////////////////////////////////////////////////////////////////////
;/ EXTERNALS
;//////////////////////////////////////////////////////////////////////////////////////////////

.extern _g_CurrentVRAMBuff      ; 16-bit pointer to the vram that is meant to be drawn
.extern _g_ColorTable           ; 16-bit pointer to the array of color tables.
.extern _g_AttribTable          ; 16-bit pointer to a int array that contains the index of the
color tables

;//////////////////////////////////////////////////////////////////////////////////////////////
;/ SRAM SECTIONS
;//////////////////////////////////////////////////////////////////////////////////////////////

; The following section describes initialized (data) near section. The reason we want
; to place these variables into the near SRAM area is so that ASM codes, that don't use many
; bits to encode data, will be able to access the variables.

.section *,data,near

_g_CurrentLine:         .int 245                        ; Line count 0->261
_g_RasterLine:          .int 0                          ; Virtual Raster count 0->191
ReptLine:               .int LINE_REPT_COUNT            ; Repeated line counter 0->2
CurrVRAMPtr:            .int 0                          ; Line count 0->525
CurrAttribPtr:          .int 0                          ; Points to g_AttributeBuf
_g_VsyncFlag:           .int 0                          ; Flag to indicate we are in v-sync
ReptAttrib:             .int 8                          ; Repeat attribute table increment 8 pixel
lines
```

This is only a small excerpt from the assembly driver and it contains all of the declarations. The externals section contains the variables that **MUST** be declared by the user’s program. These externals will be linked in to the assembled code by using their names as reference (minus the leading underscore “_”).

_g_CurrentVRAMBuff This is a 16-bit pointer to the VRAM that is to be drawn to the screen.

_g_ColorTable This is a 16-bit pointer to the color tables or as we previously called it Color Palettes or just **Palettes**. Depending on the driver (2-bit per pixel or 4) they range in size from 4 to 16 and there may be more than one.

_g_AttribTable This is a 16-bit pointer to what we previously called a **palette map**. Our palette map system is similar to the Nintendo Entertainment System (NES) graphics processor that used the name Attribute Table. We refrained from using the variable g_PaletteMap so that you might use it in your “C” code.

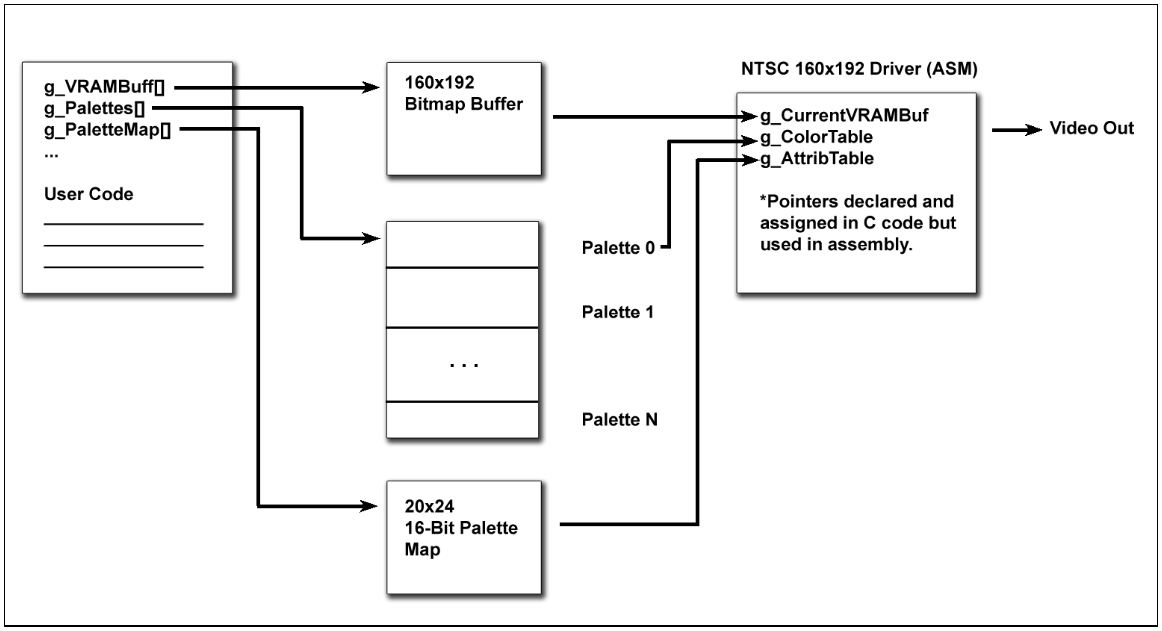
The SRAM section contains variables that are created and initialized in the assembly driver. A user may read these variables from the “C” code.

_g_CurrentLine This is a 16-bit integer that contains the current rendering line. In the case of NTSC this ranges from 0 to 261, for VGA 0 to 524.

_g_RasterLine This is a 16-bit integer that tracks the “virtual” current line of the video mode. For example, in the 160x192 mode, this variable would track from 0..191 during the active display.

_g_VsyncFlag This is a 16-bit integer that signals whether or not the video is in the Vsync/Blanking region. If it is the value is set to 0xFFFF, otherwise, if the driver is currently drawing it is cleared to 0.

Figure 15.10 – Interfacing with the 160x192 NTSC driver.



Interfacing with the driver is very simple as shown in Figure 15.10. The main C application needs to declare 3 separate globally accessible pointers and 3 separate arrays that the pointers are pointing to. The first is ***g_CurrentVRAMBuf*** which is declared as an ***unsigned char**** that points to a char array sized appropriately to fit the screen's resolution. Next we have the color table or palettes that must be defined and pointed to with an ***unsigned char**** named ***g_ColorTable***. There may be one or many color tables but they all must be contained in continuous memory. Finally there needs to be an attribute table or as previously described a palette map that is an array of unsigned shorts (16-bit) pointed to with an ***unsigned short**** named ***g_AttribTable***. A sample declaration for a 160x192x2 driver with 3 color tables is given below.

NOTE When using the XGS PIC API graphics library the variables: g_CurrentVRAMBuf, g_ColorTable, and g_AttribTable are already defined for you and you assign their values via function call.

```
// The VRAM buffer must be allocated into far memory space because it is so large it will not fit
into near
unsigned char g_VRAMBuffer[160*192/4] __attribute__((far));

// [required] A 16-bit pointer set to the current VRAM Buffer
unsigned char *g_CurrentVRAMBuf = g_VRAMBuffer;


// Our Color Tables 3 of them each 4 bytes in size in 2-bit per pixel mode
unsigned char g_Palettes[3*4];

// [required] A 16-bit pointer set to the color palettes array
unsigned char *g_ColorTable = g_Palettes;


// Table that contains which color palette to assign 8x8 group
unsigned short g_PaletteMap[20*24];

// [required] A 16-bit pointer to the current palette map
unsigned short *g_AttribTable = g_PaletteMap;
```

TIP	The PIC24 has enough RAM that it is possible to have two VRAM buffers in some graphics modes. This allows us to double buffer the video frames for certain animations or possibly to render at a slower rate giving more time for complex algorithms - 3D polygon engines anyone?

That's all there is to it. Once this data structure is allocated then ***g_CurrentLine*** and ***g_RasterLine*** can be interrogated at any time by your C code to help you keep track of where the raster is. Additionally, the palette maps and palettes themselves can be modified at anytime to change color schemes. 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 160x192 NTSC driver.
- In your C application you need to declare 3 pointer variables: ***g_CurrentVRAMBuf, g_ColorTable, g_AttribTable***. 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 palette maps, 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.

15.2.2 Plotting a Pixel Bitmapped Modes

The most important thing you need to 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 160x192 pixel mode):

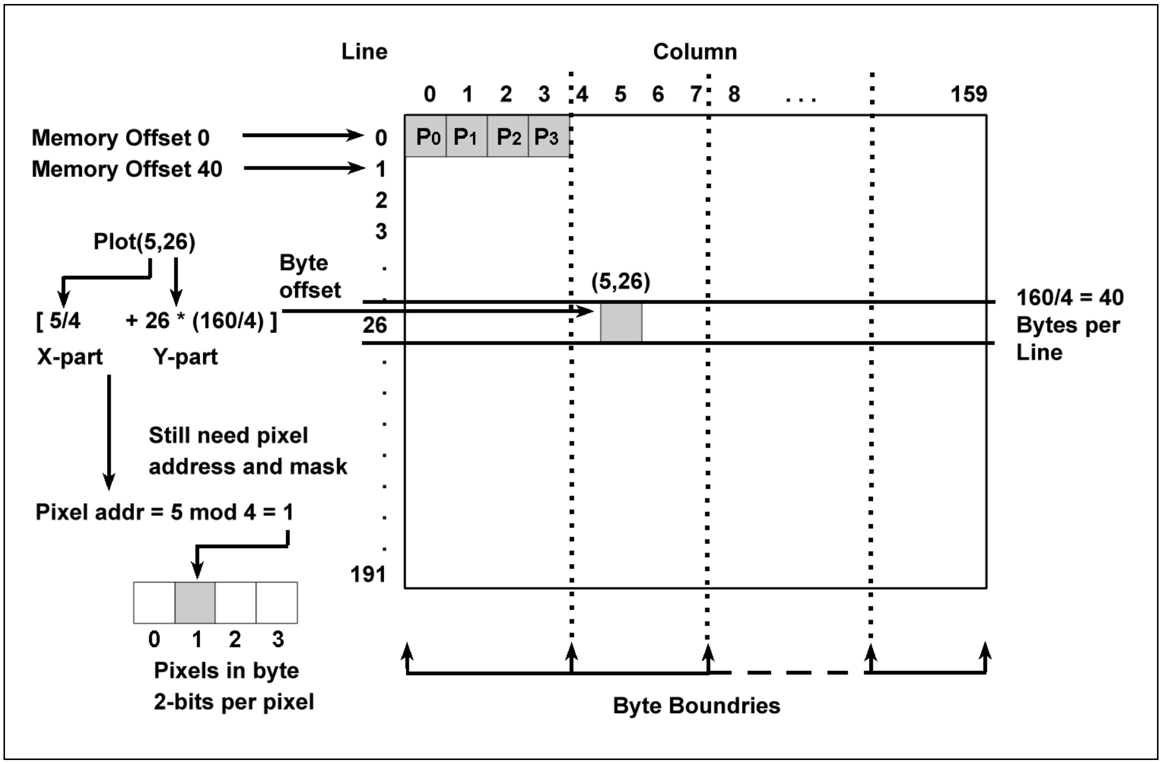
pixel_addr = y*160;

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

pixel_addr = x + y*160;

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 15.11 shows this computation for our hypothetical graphics mode with 1 byte per pixel.

Figure 15.11 – Computing pixel addresses for 1-byte per pixel graphics mode at 160x192.



However, we are using a 4 pixels per byte graphics mode that reduces our memory RAM requirement by 1/4th, but makes memory computations a little more complex. Let's start with what we know. First, the new **"memory pitch"** is **160/4**, so to compute the y contribution to the pixel address is as follows:

pixel_addr = y*160/4;

Not too terribly difficult, but what about the x contribution? It turns out that the x address must also be divided by 4 yielding us the following:

pixel_addr = x/4 + y*160/4;

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

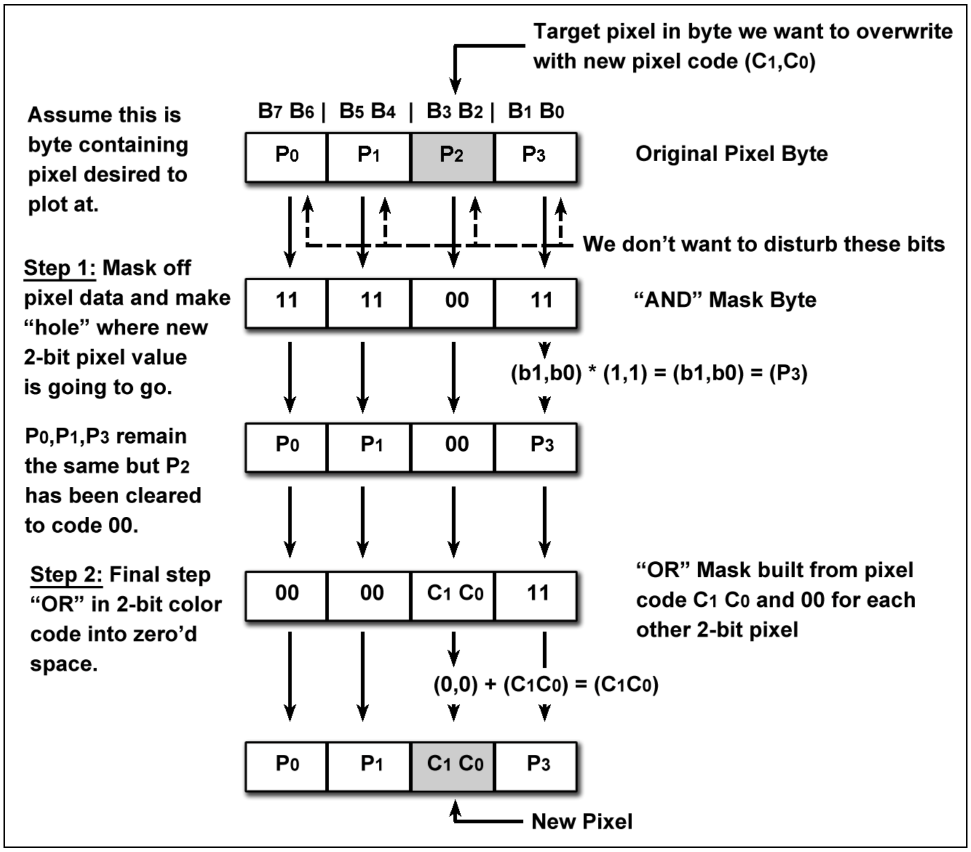
pixel_addr = (x + y*160) / 4;

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

pixel_addr = (x + y*160) >> 2;

Referring to Figure 15.12 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 15.12 – Plotting a single pixel with packed 2-bit per pixel data.


Target pixel in byte we want to overwrite with new pixel code (C1,C0)
Assume this is byte containing pixel desired to plot at.
B7 B6 | B5 B4 | B3 B2 | B1 B0
P0
P1
P2
P3
Original Pixel Byte
We don't want to disturb these bits
Step 1: Mask off pixel data and make "hole" where new 2-bit pixel value is going to go.
11
11
00
11
"AND" Mask Byte
(b1,b0) * (1,1) = (b1,b0) = (P3)
P0,P1,P3 remain the same but P2 has been cleared to code 00.
P0
P1
00
P3
Step 2: Final step "OR" in 2-bit color code into zero'd space.
00
00
C1 C0
11
"OR" Mask built from pixel code C1 C0 and 00 for each other 2-bit pixel
(0,0) + (C1C0) = (C1C0)
P0
P1
C1 C0
P3
New Pixel

Looking at the graphical representation of the algorithm in Figure 15.12, 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:

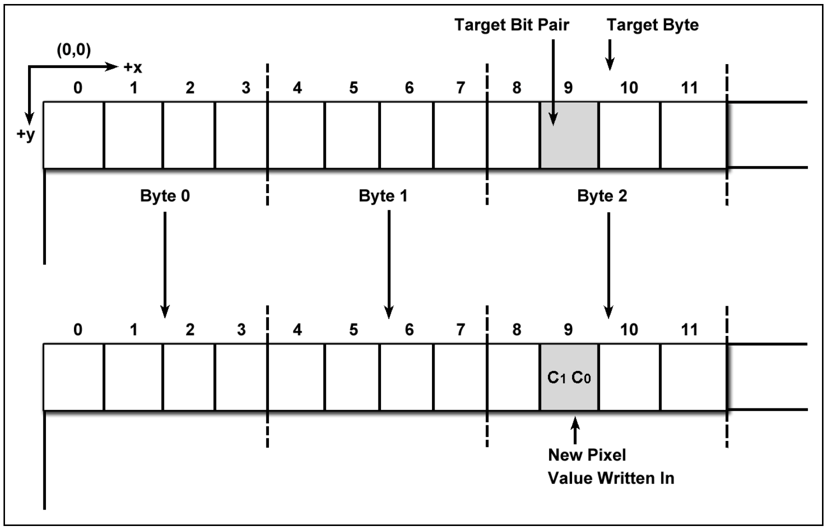
pixel_shift = 2*(x mod 4);

pixel_shift would then represent the number of times to shift **right** 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 15.13.

Figure 15.13 – 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 right resulting in the image shown in Figure 15.13 (remember we render the pixels MSB to LSB in pairs of 2). Hopefully, you see that our algorithm works. The remaining problems are with ***"masking"*** issues. Referring back to Figure 15.12, 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 performs this operation completely. One thing you will notice different from the above equations is that we are calculating the ***ShiftAmount*** variable to be the number of shifts to the left. We are doing this because it is easier to shift the passed in ***Color*** value to the left than it would be to first shift it all the way to the left and then back again to the right to fill in the hole.

```
// Plots a single pixel non-clipped to the viewport
void GFX_Plot_2BPP(int x, int y, unsigned char Color, unsigned char *VRAMPtr)
{
       // Calculate the shift amount to get to the individual pixel
       int ShiftAmount = 6 - (x & 0x03)*2;
       unsigned int Index = (g_ScreenWidth>>2)*y + (x>>2);

       // We need to mask out the previous pixel and load in the new one
       unsigned char CurValue = VRAMPtr[Index];
       CurValue &= ~(0x03 << ShiftAmount);
       CurValue |= (Color << ShiftAmount);
       VRAMPtr[Index] = CurValue;
}
```

15.2.3 Manipulating the Color Palettes

We had previously spoken about the color palettes and the palette map, however in this section we wanted to reiterate the concepts mentioned. First, all bitmap graphics modes make use of multiple color palettes to expand the number of available colors on a single screen greater than 4 for 2-bit mode and 16 for 4-bit mode. The one exception to this is the higher resolution VGA 200x200 2-bit per pixel mode. The entire screen is divided up into 8x8 pixel regions that can be individually assigned to a separate color palette. This mapping between the pixel regions and the color palettes is called the **palette map**. All bitmap drivers require that you create (globally) pointers to a continuous memory array for the color palettes and also a palette map that covers the entire screen. Each color palette is a byte array that contains the **LUMA:CHROMA** pair for NTSC televisions and the **6-bit RGB** color value for VGA displays. Since color palettes must be linear in memory you must declare the size of the char array to be a multiple of the minimum number of colors for the graphics mode. The following code shows an example of creating and setting up a *three* color palettes for 2-bit per pixel graphics (4 color).

```
// Create the color palette as a multiple size of 4 in a 2-bit graphic driver
unsigned char g_Palettes[3*4];

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

// Palette 1
g_Palettes[1*4 + 0] = color4;
g_Palettes[1*4 + 1] = color5;
g_Palettes[1*4 + 2] = color6;
g_Palettes[1*4 + 3] = color7;

// Palette 2
g_Palettes[2*4 + 0] = color1;
g_Palettes[2*4 + 1] = color4;
g_Palettes[2*4 + 2] = color3;
g_Palettes[2*4 + 3] = color7;
```

From the code above we can see that palettes can share the same colors where **color** 1,2,3,etc., represents an NTSC or VGA format.

The palette maps are then used to map the different color palettes to the screen’s 8x8 pixel regions. Expanding on the example above one might want to use the color palette 0 for the entire screen except for the first two regions. In the first and second 8x8 pixel regions the user might want to use palette 1 and 2 respectively. A snippet of code to implement this is given below.

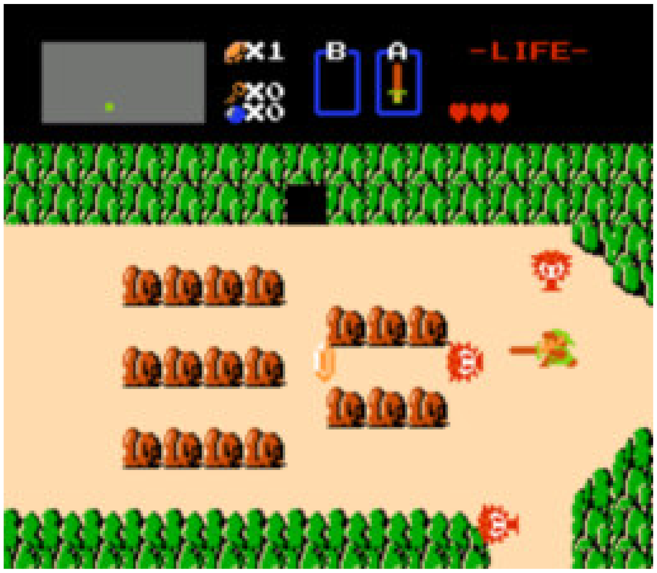
```
// The Palette map is a short array (16-bit) that contains the palette number to use
unsigned short g_PaletteMap[(SCREEN_WIDTH/8)*(SCREEN_HEIGHT/8)];

// Set the entire screen to use the first palette
for(i = 0; i < sizeof(g_PaletteMap); i++)
   g_PaletteMap[i] = 0;

// Now set the first and second regions to 1 and 2
g_PaletteMap[0] = 1;
g_PaletteMap[1] = 2;
```

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 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!

15.3 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, 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 15.14 shows one of the most popular tile mapped games and franchise in history ***"The Legend of Zelda"***.

Figure 15.14 – A screen shot of The Legend of Zelda for the NES.



Referring to the screen shot, you can immediately see the **"tiles"** in the image that are repeated. Examples are the brown rocks, the ground, the green walls, the black door, and so forth. This is the idea of tile mapping. If you look back to Figure 15.4 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 PIC, we have developed a number of tile mapped drivers for both NTSC and VGA, with varying number of sprites. 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. Listed below are the tile engines available for NTSC and VGA

NTSC Tile Engines

- XGS_PIC_NTSC_TILE1_V010.s|h — 160x192, 20x24, 8x8 tiles resolution NTSC driver tile engine with (6) 8x8 sprites.
- XGS_PIC_NTSC_TILE2_V010.s|h — 168x200, 28x25, **6**x8 tiles resolution NTSC driver tile engine with (4) 8x8 sprites.

VGA Tile Engines

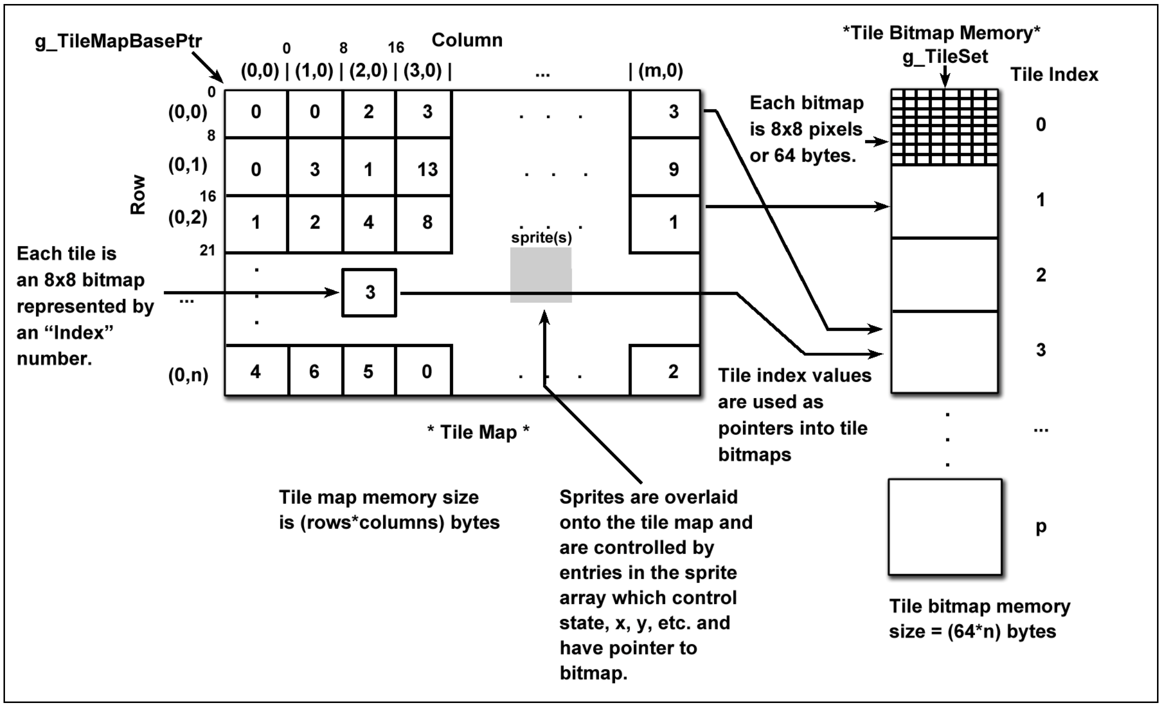
- XGS_PIC_VGA_TILE1_V010.s|h — 176x240, 22x30, 8x8 tiles resolution VGA driver tile engine with (3) 8x8 sprites.

NOTE

The DVD ROM source might have more engines on it than 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 number of sprites 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_PIC_NTSC_TILE1_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 15.15 illustrates this tile mode and annotates important details.

Figure 15.15 – A tile map mode illustrated.


g_TileMapBasePtr
0 8 16 Column
(0,0) | (1,0) | (2,0) | (3,0) | ... | (m,0)
Row
(0,0) 0
(0,1) 8
(0,2) 16
21
...
(0,n)
0 0 2 3 3
0 3 1 13 9
1 2 4 8 1
3
4 6 5 0 . . . 2
sprite(s)
Each tile is an 8x8 bitmap represented by an "Index" number.
* Tile Map *
Tile map memory size is (rows*columns) bytes
Sprites are overlaid onto the tile map and are controlled by entries in the sprite array which control state, x, y, etc. and have pointer to bitmap.
Tile Bitmap Memory
g_TileSet
Tile Index
Each bitmap is 8x8 pixels or 64 bytes.
0
1
2
3
...
p
Tile index values are used as pointers into tile bitmaps
Tile bitmap memory size = (64*n) bytes

Referring to Figure 15.15, 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 ***g_TileMapBasePtr*** and is a 16-bit pointer declared in the master graphics library module **XGS_PIC_GFX_DRV_V010.c** along with various other tile map related values, here's a glimpse:

```
volatile unsigned char *g_TileMapBasePtr = NULL;
volatile unsigned char *g_TileSet   = NULL;
```

NOTE

Often throughout this manual and the demo code you will see the keyword **volatile** used in the declaration of a variable. When using this modifier the compiler will not optimize the code that reads from or writes to the variable. This is important when an interrupt will be accessing or modifying the volatile variable since a change of this value could happen at any time. Thus the compiler has no way of knowing how the variable has been influenced.

g_TileMapBasePtr 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 unsigned char g_TileMap[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, 0, 1,    // row 0
    0, 1, 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,    // row 2
    0, 0, 0, 1, 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,    // row 4
    0, 0, 0, 0, 0, 1, 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,    // row 6
    0, 0, 0, 0, 0, 0, 0, 1, 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,    // row 8
    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,    // row 10
    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,    // row 12
    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,    // row 14
    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,    // row 16
    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,    // 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 ***g_TileMapBasePtr*** 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 ***g_TileMapBasePtr*** to the data structure like this:

```
g_TileMapBasePtr = g_TileMap;
```

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:

```
g_TileMapBasePtr += 20;
```

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

```
g_TileMapBasePtr += SCREEN_TILE_MAP_WIDTH;
```

This would have the effect of ***“scrolling”*** the image down by one tile row! 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 ***g_TileMapBasePtr*** and ***g_TileSet*** around as you wish. The later pointer we need to discuss a little more.

The second variable in the previous pointer declaration listing, ***g_TileSet*** points to the actual tile bitmaps you want used 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 unsigned char g_Tiles[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 unsigned char g_Tiles[TILE_WIDTH * TILE_HEIGHT] =
{
    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) or **BLACK** 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 with 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 statements or read right off the SD card.

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 actual RGB values are in BGR format left to right (0,0, B_1B_0, G_1G_0, R_1R_0), thus (00000011) creates a pure RED.

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

```
g_TileSet = g_Tiles;
```

This step in addition to the ***g_TileMapBasePtr*** 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 ***g_TileMapBasePtr*** and ***g_TileSet*** 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.

15.3.1 Deconstructing the Tile Map ASM Driver and the Header File

Since there are a few tile map drivers, we can't cover them all, we are going to stick with the plan and cover the tile engine driver that we have been discussing **XGS_PIC_NTSC_TILE1_V010.s|h**. The other tile map engines both NTSC and VGA are very similar, the only additions will be in relation to number of tiles and ***sprites*** 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 **"SRAM section"** from the ASM driver, it contains some additional declarations, but we can parse them out of our discussions:

```
;///////////////////////////////////////////////////////////////////////////////////////////////////
///////////////
;/ SRAM SECTIONS
;///////////////////////////////////////////////////////////////////////////////////////////////////
///////////////

; The following section describes initialized (data) near section. The reason we want
; to place these variables into the near SRAM area is so that ASM codes, that don't use many
; bits to encode data, will be able to access the variables.
.section *,data,near
_g_CurrentLine:             .int 245                   ; Line count 0->261
_g_RasterLine:              .int 0                     ; Line count 0->200
_g_TileMapLogicalWidth:     .int NUM_HORZ_TILES        ; read/write, logical width of tile map
REPTLINE:                   .int LINE_REPT_COUNT       ; Repeated line counter 0->1
TILEINDPTR:                 .int 0                     ; Current tile we are drawing
LINECOUNT:                  .int 0                     ; Line counter increments by 6 bytes 0->48
_g_VsyncFlag:               .int 0                     ; Flag to indicate we are in v-sync
_g_VscrollFine:             .int 0                     ; Fine scrolling flag 0->7

; Uninitialized data sections
.section *,bss,near
SCAN_BUFFER:   .space SCREEN_PITCH+8                  ; Contains the scanline buffer for current
line
TILE_POINTERS: .space (2*NUM_HORZ_TILES)              ; Two bytes per tile
```

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 are actually exported from the driver, they are:

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

g_TileMapLogicalWidth — 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 it 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:

```
;///////////////////////////////////////////////////////////////////////////////////////////////////
///////////////
;/ EXTERNALS
;///////////////////////////////////////////////////////////////////////////////////////////////////
///////////////

.extern _g_TileMapBasePtr       ; 16-bit pointer to tile map, allows indirection
.extern _g_SpriteList           ; 16-bit pointer to sprite table records
.extern _g_TileSet              ; 16-bit pointer to 8x8 tile bitmaps, allows indirection
.extern _g_ColorTable           ; 16-bit pointer to a Color Table for pixel color indirection

;///////////////////////////////////////////////////////////////////////////////////////////////////
///////////////
```

```
;/ GLOBAL EXPORTS
;///////////////////////////////////////////////////////////////////////////////////////
///////////////

.global __T1Interrupt           ; Declare Timer 1 ISR name global
.global _g_VsyncFlag            ; int, read, Flag to indicate we are in v-sync
.global _g_VscrollFine           ; int, read/write, Fine scrolling flag 0->7
.global _g_CurrentLine          ; int, read only, current physical line 0...524
.global _g_RasterLine           ; int, read only, current logical raster line
.global _g_TileMapLogicalWidth  ; int, read/write, logical width of tile map
```

The driver imports ***g_TileMapBasePtr***, ***g_TileSet, g_SpriteList, and g_ColorTable*** and 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, sprites, and color table, but the other variables the driver declares and external modules can link with them and use them. Again notice the ***g_VsyncFlag, g_CurrentLine*** and ***g_RasterLine*** 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_PIC_NTSC_TILE1_V010.h** starts off with a few useful macros:

```
// simply interrogate the video driver's g_CurrentLine variable and determine where the scanline
is
#define VIDEO_ACTIVE(video_line)          ( ((video_line) >= START_ACTIVE_SCAN && (video_line) <
                                          END_ACTIVE_SCAN))
#define VIDEO_INACTIVE(video_line)        ( ((video_line) < START_ACTIVE_SCAN || (video_line) >=
                                          END_ACTIVE_SCAN))
#define VIDEO_TOP_OVERSCAN(video_line)    ( ((video_line) >= 0 && (video_line) <
START_ACTIVE_SCAN))
#define VIDEO_BOT_OVERSCAN(video_line)    ( ((video_line) >= END_ACTIVE_SCAN))
```

These ugly macros let you interrogate the raster line and determine if it's 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 ***g_CurrentLine*** 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.

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

```
// Common NTSC colors
#define COLOR_LUMA     (0x91)

#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 + 13)

#define NTSC_BLACK     (0x4F)
#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.

```
// NTSC screen size
#define SCREEN_WIDTH          160
#define SCREEN_HEIGHT         192
#define SCREEN_BPP            8

// Starting position on screen
#define VERTICAL_OFFSET       40
#define LINE_REPT_COUNT       1

// 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*LINE_REPT_COUNT + VERTICAL_OFFSET)
#define END_VIDEO_SCAN       261
```

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

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:

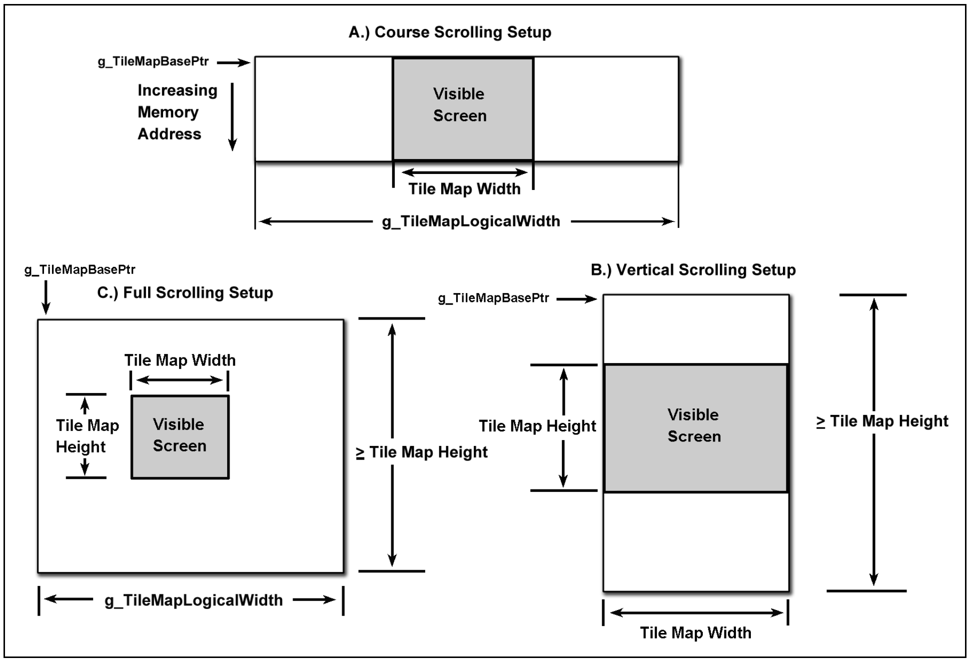
```
// Used for allocating buffers in main program
#define MAX_PALETTE_SIZE              256

#define SCREEN_TILE_MAP_WIDTH         20
#define SCREEN_TILE_MAP_HEIGHT        24
#define TILE_WIDTH                    8
#define TILE_HEIGHT                   8

#define MAX_SPRITES                   6
```

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

15.3.2 Scrolling Tile Maps

In the previous discussions we have touched upon 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 ***g_TileMapBasePtr*** 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 ***g_TileMapLogicalWidth*** variable comes in. Normally, you would set ***g_TileMapLogicalWidth*** to the tile map width itself ***SCREEN_TILE_MAP_WIDTH***, but say you wanted to have 10 screens of data horizontally then you would set the variable to ***10*SCREEN_TILE_MAP_WIDITH***. 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 ***g_TileMapLogicalWidth***, and then for all intent purposes you have horizontal course scrolling. Figure 15.16 illustrates both setups for horizontal and vertical course scrolling.

Figure 15.16 – Course scrolling for horizontal and vertical setups.

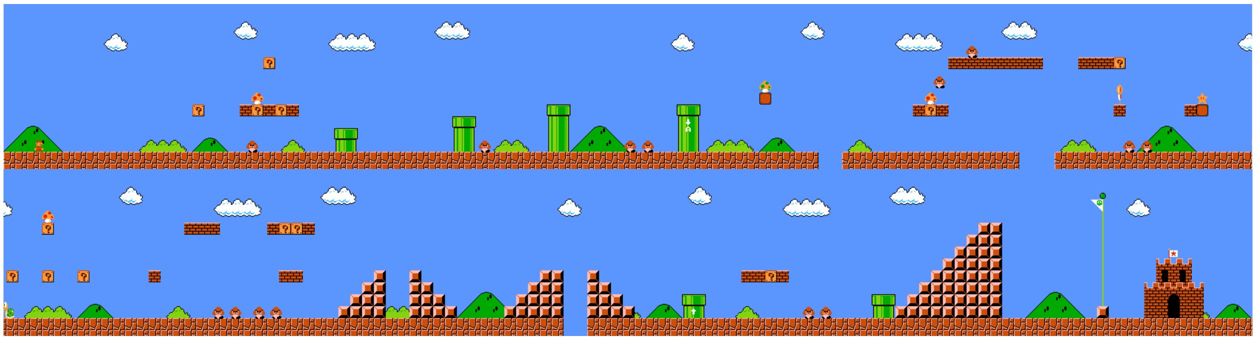


Thus to support vertical course scrolling you need to add rows to your tile map, set the logical width to the standard size of the mode and then adjust the ***g_TileMapBasePtr*** by the memory pitch per row to scroll up/down. For example, the pitch would be 20 for a 20x24 tile mapped mode, we have seen 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 ***g_TileMapLogicalWidth*** variable, so the driver knows how wide you want to interpret the tile map data as. Then you once again adjust the ***g_TileMapBasePtr***, but this time by +-1 if you want to scroll left or 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 unsigned char g_TileMap[ 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 0,    // row 23
};
```

As an example of how far you can take this, take a look at Figure 15.17 below, it shows the entire game world layout for the ***"Super Mario Brothers"*** (stacked on each other to fit on the page better). But, this game is only horizontally scrolled during gameplay as you know if you have played it.

Figure 15.17 – 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 do 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 ***g_VscrollFine***. 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 is the following algorithm:

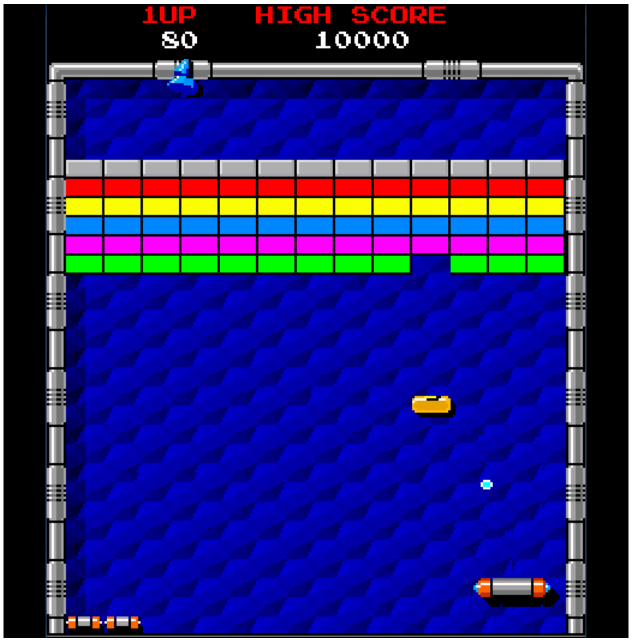
```
if (++g_VscrollFine > 7)
{
   // reset fine scroll
   g_VscrollFine = 0;

   // course scroll one line
   g_TileMapBasePtr += g_TileMapLogicalWidth;

} // end if
```

Notice I used ***g_TileMapLogicalWidth*** instead of ***SCREEN_TILE_MAP_WIDTH***, this is a good tactic since this is always setup (you set it and its defaulted in the ASM driver to the tile map width) 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.

15.3.3 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 indices.

Figure 15.18 – The classic game "Arkanoid".



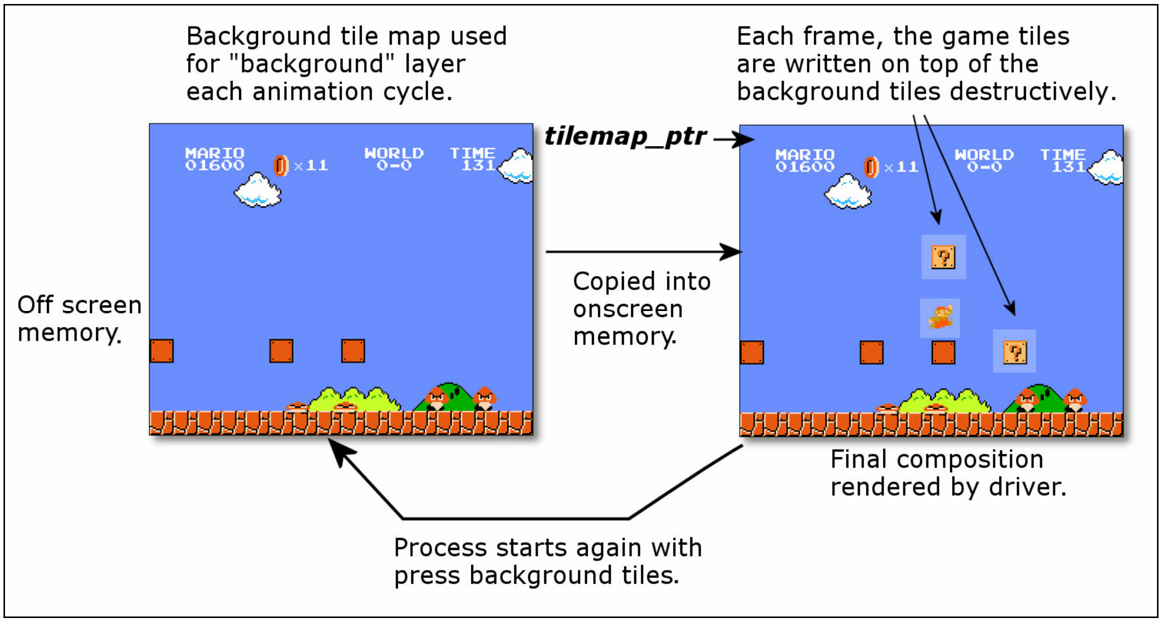
For example, say that you had written a breakout clone game like **"Arkanoid"** shown in Figure 15.18. This game is tile mapped of course and the breakout blocks are tiles. As the ball bounces around the screen and collides with one of the colored bricks, you need to replace it with the background tile. The code might look like this:

```
tile_map[ brick_tile_x + brick_tile_y*SCREEN_TILE_MAP_WIDTH ] = background_tile;
```

So to animate any tile you simply write into the tile map 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 cannot 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.

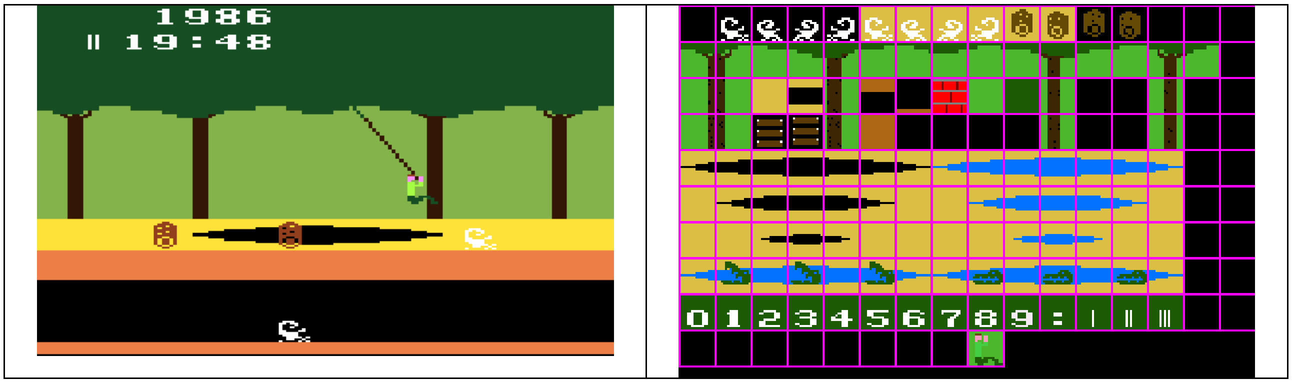
Let's address the first problem; the loss of data and potential storage of the data. There are a couple ways to approach this problem. One method is to build a new tile map every single frame of animation. That is every 30th or 60th of second (whatever frame rate you run the animation at), you actually 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 15.19 shows this graphically.

Figure 15.19 – 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 15.20.

Figure 15.20 – 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*SCREEN_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*SCREEN_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* SCREEN_TILE_MAP_WIDTH];

// now overwrite that tile with the scorpion
tile_ptr[ scorp_x + scorp_y* SCREEN_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.

15.3.4 Tile Engines and Sprite Support

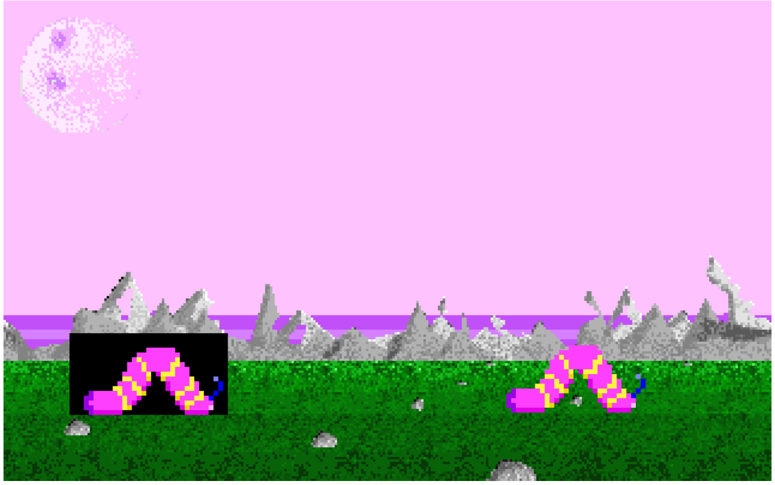
Sprites, as mentioned before, augment tile engines by allowing a fixed number of ***“overlayed”*** bitmaps to move freely around the tile map display without disturbing the tile map itself. Sprites are typically drawn after the tile map or on top of it either with software algorithms or physical hardware 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 support scaling operations in the future. 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 PIC tile engine drivers.

First, just to make sure we get what a sprite is again, take a look at the “Arkanoid” screen shot in Figure 15.18, the background is made up of tiles, but the foreground objects like the paddle, falling powerups, and the ball are sprites. Notice how they overlap the tiles (background tiles in this case) and can move around anywhere on the screen? If these objects were not sprites, this would be 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 in advanced engines - 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 15.21 shows and example of transparency and opaque background.

Figure 15.21 – Transparent and non-transparent sprite rendering.



As you can see, the transparent sprite on the right is 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 of 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 sprites are always drawn as 1x and at this time they do not support scaling. In the future this may change so make sure you read the header file information on the engines to find out their exact features and limitations.

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, it’s 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_PIC_VGA_TILE1_V010.s**, this is a VGA tile driver with the following specs:

- 176x240 pixel resolution.
- 22x30 tiles each 8x8 in size.
- 3 sprites with transparency.

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).

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 = 175 places the sprite beyond the right edge of the screen. In other words, the sprite coordinates look like the diagram shown in Figure 15.22.

Figure 15.22 - Sprite coordinates relative to the tile map.

```
        <------- visible on screen --------->|, x's represent tile imagery

        0                                    175        <--- tile pixels (22*8 = 0...176)
        |                                     |
--------xxxxxxxxxxxxxxxxxxxxxxx....xxxxxxxxxxxx--------
|      |                                       |
0      7                                       183      <--- sprite X coordinates (0..183)
```

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 _g_TileMapBasePtr    ; 16-bit pointer to tile map, allows indirection
.extern _g_TileSet           ; 16-bit pointer to 8x8 tile bitmaps, allows indirection
.extern _g_SpriteList        ; 16-bit pointer to sprite table records
```

First, are the tile map, tile bitmap pointers which we have seen 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 above pointers at them, so the tile engine could locate them.

Now, the new data structure is a 16-bit pointer that points to the sprite table base address, thus the caller defines the ***g_SpriteList*** and points it to an array of sprite records, where each record is of the type:

```
// The following is the sprite structure
typedef struct{
        unsigned char State;                 // State of the Sprite (SPRITE_STATE_ON/OFF)
        unsigned char Attribute;             // Attribute of the sprite (NOT USED)
        unsigned int X;                      // X position of the sprite
        unsigned int Y;                      // Y position of the sprite
```

```
       unsigned char *PixelData;             // Pointer to the bitmap data of the sprite
} Sprite, *SpritePtr;
```

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

In most incarnations of the tile engines only State, X, Y, and *PixelData are used. The Attribute field is for the caller's own use. Other than that the only other interesting field is the bitmap pointer ***PixelData***. 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, thus 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 a 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***, 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 unsigned char sprite_bitmap[ TILE_WIDTH*TILE_HEIGHT*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. You might have noticed that this definition is larger than an 8x8 tile. What you can do is chain 4 different sprites together to create a larger 8x32 sized 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_list[NUM_SPRITES] = {
                                             {SPRITE_ON,0,20,30,sprite_bitmap},    // sprite 0
                                             {SPRITE_ON,0,60,50,sprite_bitmap+64}, // sprite 1
                                             {SPRITE_OFF,0,80,60,sprite_bitmap}    // sprite 2
                                            };
```

In this case, sprite 0 is enabled (visible) and its (x,y) is (20,30) and the bitmap data starts at ***&sprite_bitmap***, sprite 1 is also enabled and has (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 it is initially not visible when the screen is rendered.

In conclusion, sprites add a lot of 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.

15.3.5 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 twofold; 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.

15.4 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. 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.

Join me and together we can rule the galaxy…

Well, you made it! If you got this far then you should be a master at the XGS PIC 16-Bit and hopefully see the true power of the Microchip 16-bit series processors and in general have learned to apply these techniques to any processor that you might work within your embedded systems development.

I’ve been overly impressed with the PIC 16-bit processors since starting this project. From the user interface that MPLAB presents to the simple but powerful assembly language instructions, the experience has been worthwhile for me and I hope the same is said for you. I used to think Atmel AVR all the way baby, but I think I might be starting to join the dark side.

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 PIC programmers. Also, if you have any questions, comments, or remarks please email us at **support@nurve.net**.