Driving A VGA Display With A Cypress PSoC

Summary

This document describes how to use a Cypress CY8C29466 Programmable System-on-Chip (PSoC) to drive a standard VGA display. Minimal external components are required to accomplish this feat. Horizontal and vertical sync pulse generation, video blank timing, pixel serialization, and the generation of three-bit foreground and background colors are implemented via ten PSoC digital blocks. External communications are accomplished through two PSoC UART blocks. The driver is realized in a PSoC Designer project, which accompanies this document.

Introduction

This project was started on a whim. One day, I was perusing the Internet, looking at different projects using microcontrollers. I came across a project that used a microcontroller to drive a VGA display. The rationale for the project was that VGA displays are pretty standard and are readily available. With the switch over to LCD-based XVGA displays, CRT-based VGA and SVGA displays are available at very low cost, if not for free. Indeed, the disposal of these displays in landfills, rather than properly recycling them, is becoming a significant environmental issue.

As I continued my investigations, I found that most projects of this type actually had the microcontroller driving an NTSC or PAL television display. The few examples I found where a microcontroller drives a VGA display were static demonstrations or dynamically-rendered artistic endeavors for the demoscene community. I could understand the reason for the lack of usable VGA display drivers versus television display drivers. First, the timing requirements for VGA display are more strict compared to NTSC and PAL. Second, small and inexpensive televisions are readily available.

At any rate, I became curious about whether a Cypress PSoC device could be used to drive a VGA display. I have a Cypress CY3210 PSoCEval1 board with the CY8C29466 PSoC. It contains sixteen digital blocks and twelve analog blocks. I had a hunch that it could successfully drive a VGA display if I cleverly employed those blocks to handle the critical timing requirements of a VGA display.

The following section describes VGA timing requirements and how they are addressed using PSoC modules. The next section describes the PSoC modules and support functions used to perform pixel serialization, video blanking, and generation of three-bit foreground and background colors. After that, serial data communication PSoC modules and the external hardware used are described. Finally, the firmware and memory used to control the VGA driver are described.

Implementing VGA Timing

First thing I discovered while investigating the VGA standard is that there really is no VGA standard. Instead, there are a number of general timing guidelines that specify different screen resolutions. My primary source of information is the section entitled *Necessary timing information about VGA modes* found on the *VGA timing information* web page at ePanorama.net[1]. The horizontal line and vertical frame timing for "industry standard VGA" are summarized in the timing diagrams shown in Illustration 1 and Illustration 2, respectively. Note that horizontal timings are stated in terms of dot clocks, while vertical timings are in terms of horizontal lines.

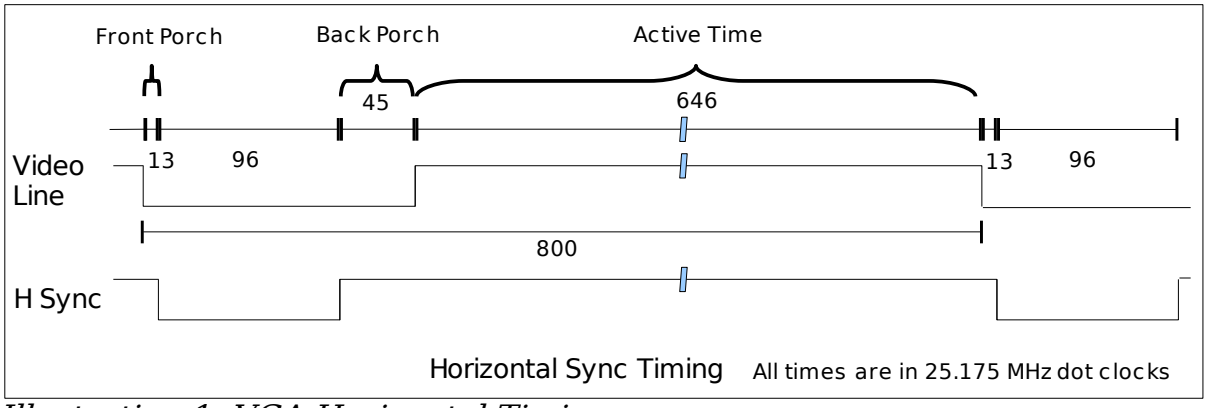

Illustration 1: VGA Horizontal Timing.

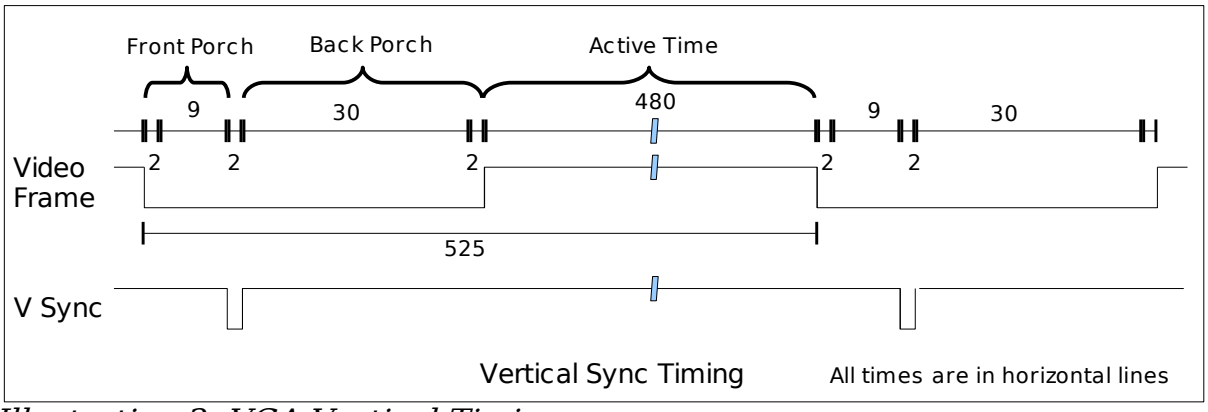

Illustration 2: VGA Vertical Timing.

For the standard VGA resolution of 640 pixels per line horizontally and 480 lines vertically, the horizontal dot clock has a frequency of 25.175 MHz. There are 800 dot clocks for each horizontal line. This includes the active video time (646 dot clocks), the horizontal front porch (13 dot clocks), the horizontal sync pulse (96 dot clocks), and the horizontal back porch (45 dot clocks). During the active video time, each dot clock period represents the rendering of one pixel (640 visible pixels plus six additional overscan/border pixels). The horizontal sync signal is a TTL-level signal sent to the VGA display. The video line exemplifies one of three video signal channels, one each for the red, green, and blue colors for the display. During the active video time, each video line is modulated with a signal representing the intensity in that color channel for each pixel. The voltage ranges between 0.0 Volts and 0.7 Volts. During the horizontal blanking period, represented by the horizontal front porch, horizontal back porch, and horizontal sync, each channel is set to 0.0 volts.

1http://www.epanorama.net/documents/pc/vga_timing.html

For VGA vertical timing, each frame consists of 525 horizontal lines. Of those horizontal lines, 480 are for active video display. The remaining lines are divided between the vertical front porch (9 lines plus 2 overscan/border lines), the vertical sync pulse (2 lines), and the vertical back porch (30 lines plus 2 overscan/border lines). Only during the active video frame time can the red, green, and blue channels be driven, and only, then, during the active video time. The vertical sync signal is a TTL-level signal sent to the VGA display.

Using the standard dot clock of 25.175 MHz, each VGA horizontal line takes 31.778 microseconds, for a horizontal scan frequency of 31469 Hz. Each VGA frame then takes 16.883 milliseconds, for a vertical scan frequency of 59.94 Hz. This is where the standard VGA monitor's horizontal timing of 31.5 kHz and vertical timing of 60 Hz come from. The horizontal sync pulse width is 3.813 microseconds long and the vertical sync pulse width is 63.56 microseconds long, both being negative-going pulses.

From studying the aforementioned eParanorama.net web page, it can be seen that these timings are not rigidly defined. There are allowances for variations in the VGA display timings. Most VGA displays are designed to synchronize and display a picture even with deviations in the active video timing and the horizontal and vertical sync frequencies and pulse widths, as long as the deviations are not too severe. The screen height, width, and scale controls found on monitors enable the user to adjust to compensate for these deviations. The PSoC-based VGA display drive will take advantage of these allowances.

Two 16-bit PWM dead band generators (PWMDB16) will be used to generate the horizontal and vertical sync signals and their corresponding active video time and active frame time signals. The advantage of these user modules is that only three digital blocks are needed to generate two synchronous and complementary signals. The VGA sync signals are active when their active time in not active, and vice versa.

The timing diagram for a PWMDB16 user module[2] is shown in Illustration 3. Note that the signals Phase 1 and Phase 2 are synchronous and complementary signals. The dead band generator function places a inactive gap between the active high regions of the two phases. Now, imagine if these two phases were used as VGA horizontal or vertical timing signals. Think of Phase 1 as either the active video time of the horizontal timing or the active frame time of the vertical timing. Then, if Phase 2 were inverted, it could serve as either the horizontal sync or the vertical sync, respectively. Careful selection of the PWMDM16 generator periods, pulse widths, and dead band times will generate appropriate front porch and back porch times and sync pulse widths.

The horizontal sync generator (HSYNC) user module is driven by an external 10.000 MHz crystal oscillator. While the 24 MHz internal oscillator of the PSoC is probably stable enough for sync generation, the external oscillator will also be used to generate the 5 MHz dot clock. Note, also, that 10 MHz cannot be easily derived from the 24 MHz internal oscillator. The period of the HSYNC user module is 318 clocks, the pulse width is 269 clocks, and the dead band time of 11 clocks. This yields an active video time of 258 clocks (25.8 microseconds) and a horizontal sync time of 38 clocks (3.8 microseconds). The unusual length of the active video time will be explained later. The timing diagram for the HSYNC user module is shown in Illustration 4.

The vertical sync generator (VSYNC) user module is clocked by Phase 1 of the HSYNC

2 8- and 16-Bit PWM Dead Band Generators, PWMDB v2.4, Cypress Microsystems.

user module. To support VGA vertical timing, the period of the VSYNC user module is 525 clocks. The pulse width is 502 clocks and the dead band time is 21 clocks. This yields an active frame time of 481 clocks (lines) and a vertical sync time of 2 clocks (lines). While 481 lines seems unusual, there will actually be only 480 video lines displayed per frame. The timing diagram for the VSYNC user module is shown in Illustration 5.

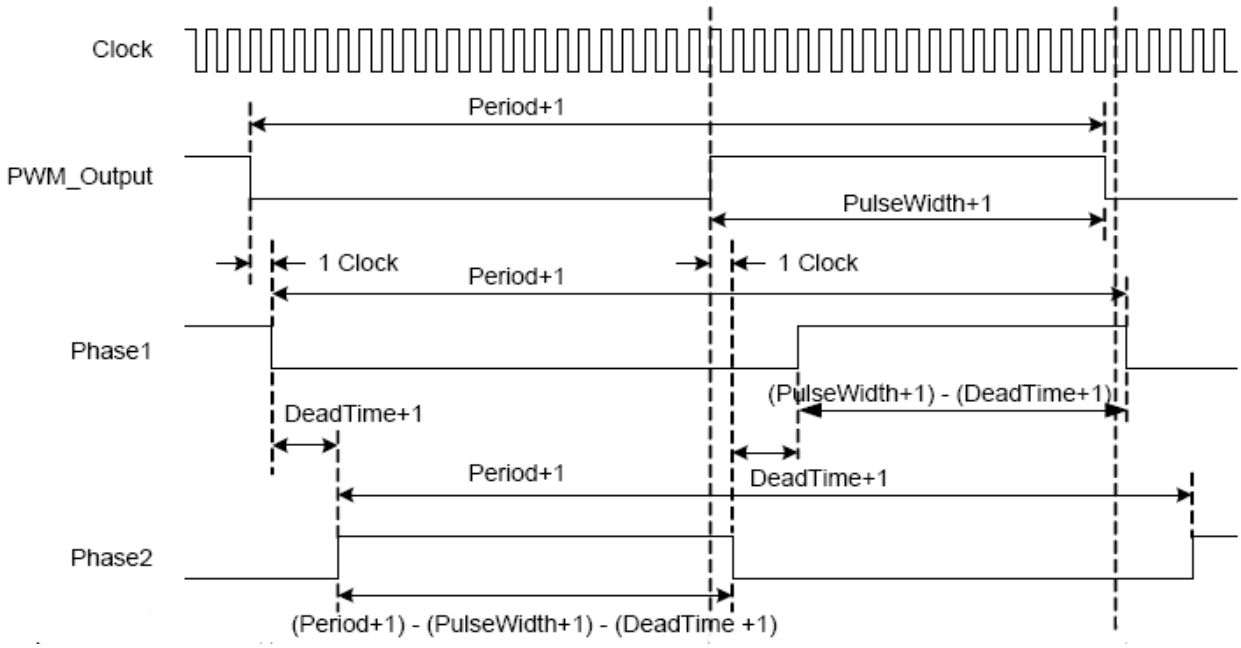

Illustration 3: Timing Diagram for the 16-Bit PWM Dead Band Generator

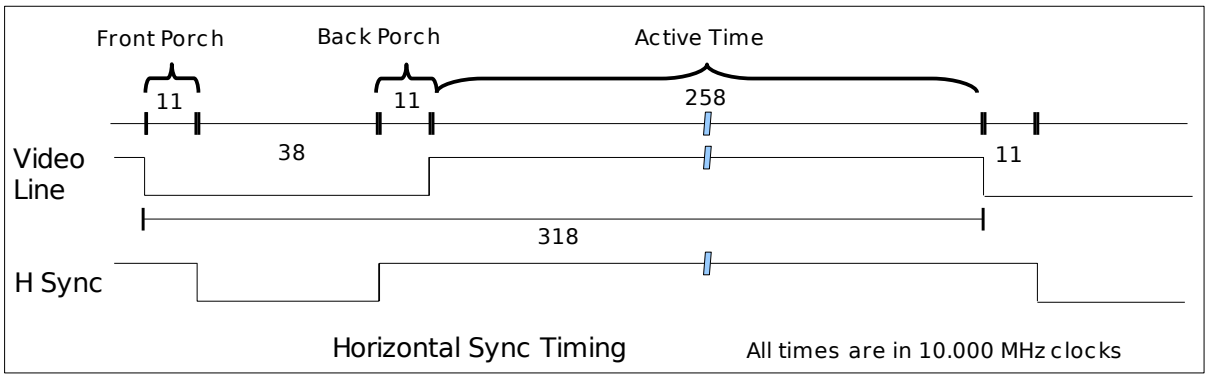

Illustration 4: HSYNC (PWMDB16) User Module Timing.

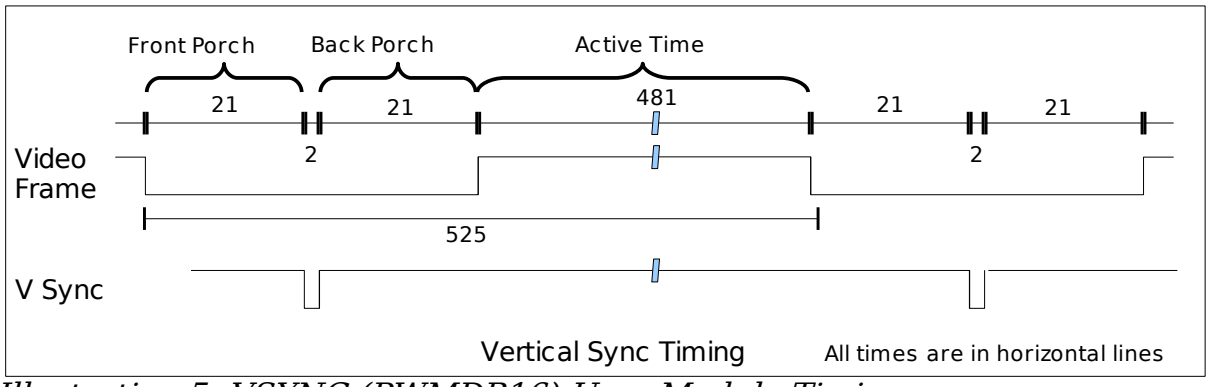

Illustration 5: VSYNC (PWMDB16) User Module Timing.

The horizontal sync signal (Phase 2 of the HSYNC user module) is inverted using a digital row LUT and output on line 0 of Port 1 (_HSYNC). The non-inverted signal is also gated with the output of a line masking (LineMask) user module via a digital row LUT to implement a video blanking signal (described below). The horizontal active video time signal (Phase 1 of the HSYNC user module) is ANDed with the video blanking signal and output on line 2 of Port 1 (HV_TIME). The signal is also ANDed with the 10 MHz clock to provide an intermittent clock signal for pixel serialization (described below). The signal is also routed via the broadcast bus to provide a clock signal to the VSYNC user module.

The vertical sync signal (Phase 2 of the VSYNC user module) is inverted using a digital row LUT and output on line 1 of Port 1 (_VSYNC). The vertical active frame time signal (Phase 1 of the VSYNC user module) is output unmodified on line 3 of Port 1 (VF_TIME). The VF_TIME signal is also directed back into the GDI to act as an interrupt trigger on its rising edge.

Pixel Serialization, Video Blanking, and Color Generation

The pixel data is stored as eight, one-bit pixels in a byte in the video raster memory. During screen rendering, the pixel bytes need to be serialized into a bit stream that is used to drive the color channels. Pixels are serialized via an SPIM user module (PixelData). The clock for the SPIM user module is an intermittent signal derived from the ANDing of the 10 MHz clock and the horizontal active video time signal.

During the horizontal blanking time, the SPIM is not clocked, so no data is being transferred out of it. At this time, the SPIM module can be stopped, restarted, and loaded with the first byte of the new video line without the transfer actually starting. Once the active video time starts, the first two clocks received by the SPIM module initialize the data transfer and load the data in the SPIM transmit buffer register into the SPIM shifter. Subsequent clocks transmit the byte out of the shifter (two clocks per bit, hence a 5 MHz dot clock). Firmware ensures that the transmit buffer is kept full so there is a constant flow of pixel data to the display during the active video time. The number of clocks necessary to serialize N bytes of data is given by the relationship

$$clocks = (16 * N) + 2 \quad .$$

For a video line containing 128 one-bit pixels, those pixels can be packed into 16 bytes. It would then take 258 clocks to serialize all of the bytes, during the active video time. Note that this corresponds to the 258-clock horizontal active video time described previously. The output of the SPIM module is routed via the broadcast bus to two non-inverting digital buffers (VGA_Buf1 and VGA_Buf2). These signals are combined with the video blanking signal by the color generation function (described below).

The video blanking function guarantees that the video channel signals are forced to 0.0 Volts during each horizontal scan when pixels are not being sent to the display. The horizontal active video signal generated by the HSYNC user module is actually longer than it needs to be. As mentioned above, the serialization of 16 bytes requires 258 clocks. Ideally, the horizontal active video time should exactly match the number of pixels transferred, i.e. 256 clocks. If the leading two clock periods could be masked out, then the proper horizontal active video time would be obtained.

The configuration and interconnection of the HSYNC and LineMask user modules

shown in Illustration 6 implement the video blanking function. The LineMask (CNTR8) user module is set up to count as long as the enable line is low. The enable signal is the output of the digital row LUT on RO0[3], which implements an A AND NOT B function. The A input of the LUT is the compare output of the CNTR8 user module. The B input is the Phase 2 output of the HSYNC (PWM16DB) user module. As long as the compare output is low or the Phase 2 output is high, the enable signal will be low and the counter will count.

The CNTR8 period is set to 50 (51 clocks), with a compare value of 0. When the Phase 2 output is low and the counter activates the compare output, the AND gate output goes high, causing the counter to stop counting. When the Phase 2 output goes low, the AND gate output goes low, allowing the counter to reload the count period. The compare output also goes low, ensuring that the counter will keep counting even when the Phase 2 output goes low again. The CNTR8 period clocks is the sum of the Phase 1 high time (38 clocks), the back porch time (11 clocks), and the first two clocks of the horizontal active video time (for 51 clocks total). The ANDing of the CNTR8 compare out with the horizontal active video time (Phase 1) signal masks off the first two clocks yielding a video blanking signal of exactly 256 clocks. The video blanking signal is properly aligned with the serialized pixel signal, so ANDing these signals will guarantee that the video channels will be forced to 0.0 Volts during the horizontal video blanking period. The combining of these signals via the Color Generation function is described below.

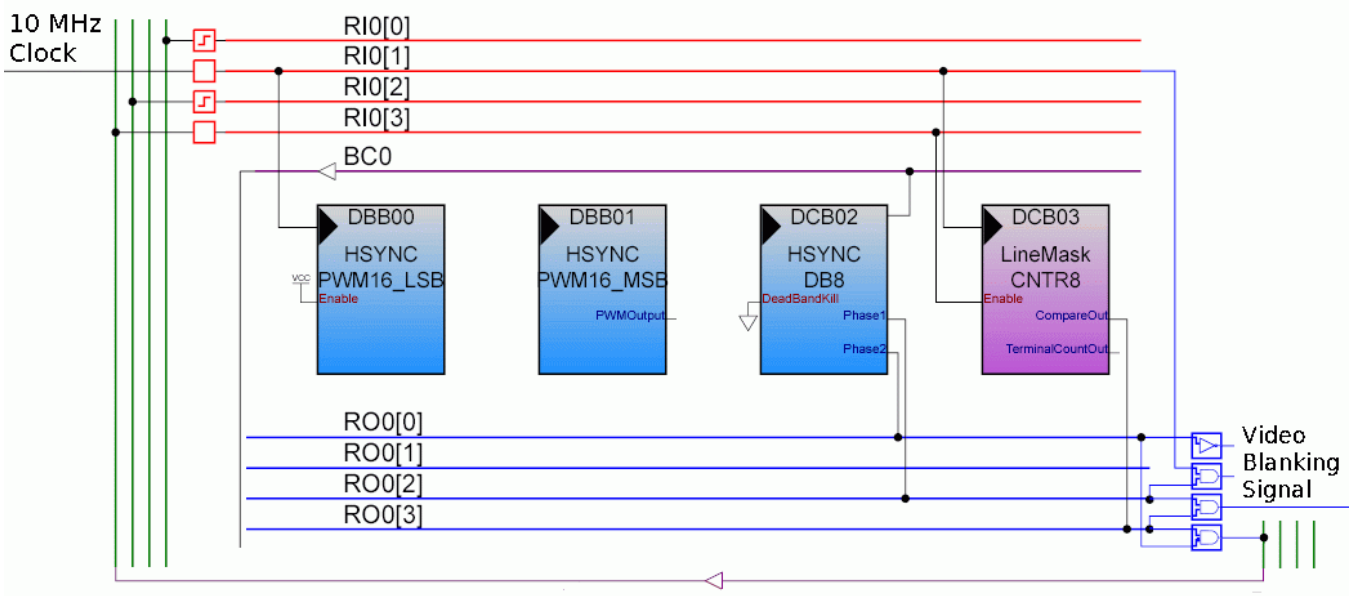

Illustration 6: Implementation of the Video Blanking Function.

The Color Generation function consists of three digital row LUTs. Each LUT combines the serialized bit stream output by the PixelData (SPIM) user module with the horizontal video blanking signal. These signals are distributed to the three LUT via two DigBuf user modules (VGA_Buf1 and VGA_Buf2). The user modules and signal interconnection are shown in Illustration 7. The serialized bit stream is connected to the digital buffers via the broadcast bus. The video blanking signal is connected to the digital buffers through a digital row input.

The outputs of the three digital row LUTs are directed through the global bus to lines 4, 6, and 7 of Port 1 to provide signals for the red, green, and blue video channels, respectively. The binary nature of these signals cause the video signals to be either at

their minimum value or their maximum value. Consequently, eight colors can be formed by the color generation function. These colors are listed in Table 1.

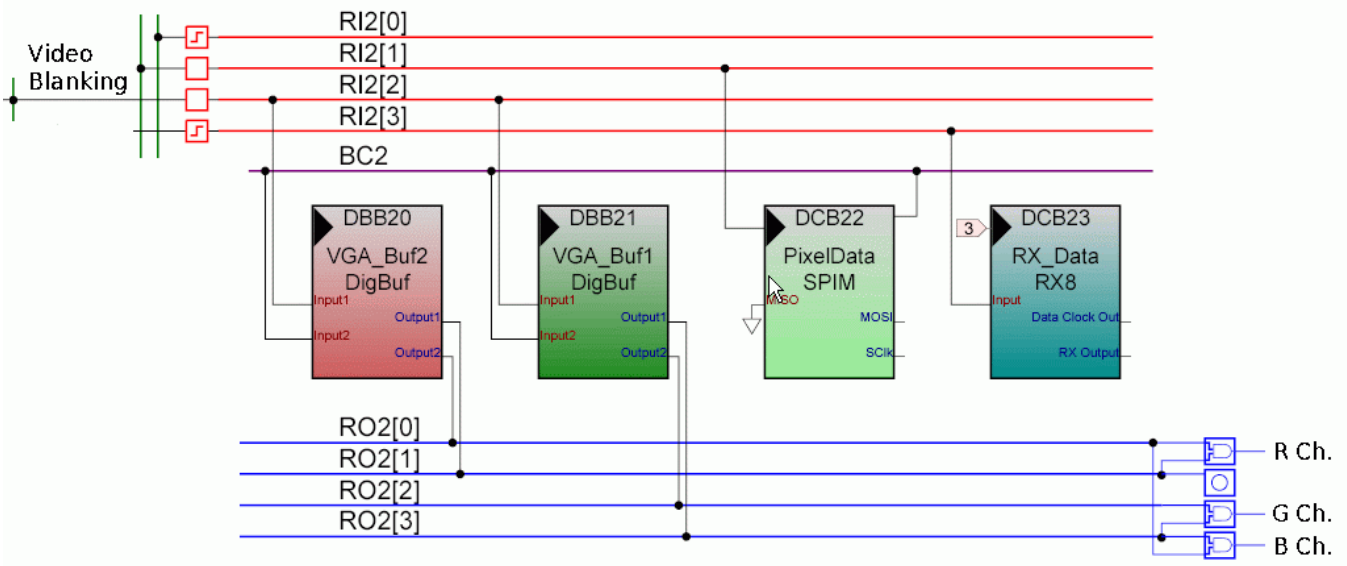

Illustration 7: Color Generation Function.

B	G	R	Color
0	0	0	Black
0	0	1	Red
0	1	0	Green
0	1	1	Yellow
1	0	0	Blue
1	0	1	Magenta
1	1	0	Cyan
1	1	1	White

Table 1: RGB Color Derived from Three-Bit Combinations

The logical function of each digital row LUT can be set to form an arbitrary operation on its inputs A and B. For the Color Generation function, specific logical functions are selected such that either the non-inverted or the inverted serialized pixel data is ANDed with the horizontal video blanking signal. In addition, the serialized pixel data can be ignored entirely, with the LUT set to output 0 or 1 during the horizontal active video time. Judicious configuration of the three LUTs results in specific foreground and background colors for the pixel states of 1 and 0, respectively. Sixty-four color combinations can be obtained, although eight of those combinations yield identical foreground and background colors. These eight combinations still have utility, as they can be used to blank the VGA display without affecting the display raster.

If the three digital row LUTs are statically set, the VGA display will show only a single foreground and background color combination across the entire screen. Multiple foreground/background color combinations can be displayed by using firmware to change the configuration of the LUTs at appropriate times. This technique is described

in the section of this document describing the firmware.

Serial Communication and External Hardware

For demonstration purposes, command and control of the PSoC VGA Display Driver is performed through a RS-232 serial interface. An I2C or SPI bus interface would work as well. An RS-232 serial interface was chosen because the Cypress CY3210 PSoCEval1 board has an Maxim MAX3232 RS-232 driver/receiver and DE-9 connector as part of its circuitry. This type of interface also makes it easy to test the PSoC VGA Display Driver with a computer that has a serial port.

An RX8 user module (RX_Data) is set up for data reception and is connected through the GDI bus to line 7 of Port 2 (RX_Data). A TX8 user module (TX_Data) is set up for data transmission and is connected through the GDI bus to line 6 of Port 2 (TX_Data). The full duplex serial data channel is set to 115200 bps. The VC3 source is set to use the 24 MHz SysClk and the VC3 divider is set to 26. This produces a VC3 clock frequency of about 923077 Hz, which yields an actual baud rate of 115384.6 bits per second. The baud rate error, compared to 115200 bps, is about 0.16%, which is acceptable.

External components required for the PSoC VGA Display Driver are minimal. All of its functionality is contained within the CY8C29466 PSoC. The essential interconnections between the PSoC and the external components are shown in Illustration 8. The pins of the CY8C29466 are color coded to show the type of internal connection for each pin. Note that most of the pins of the PSoC device (colored blue) are not used.

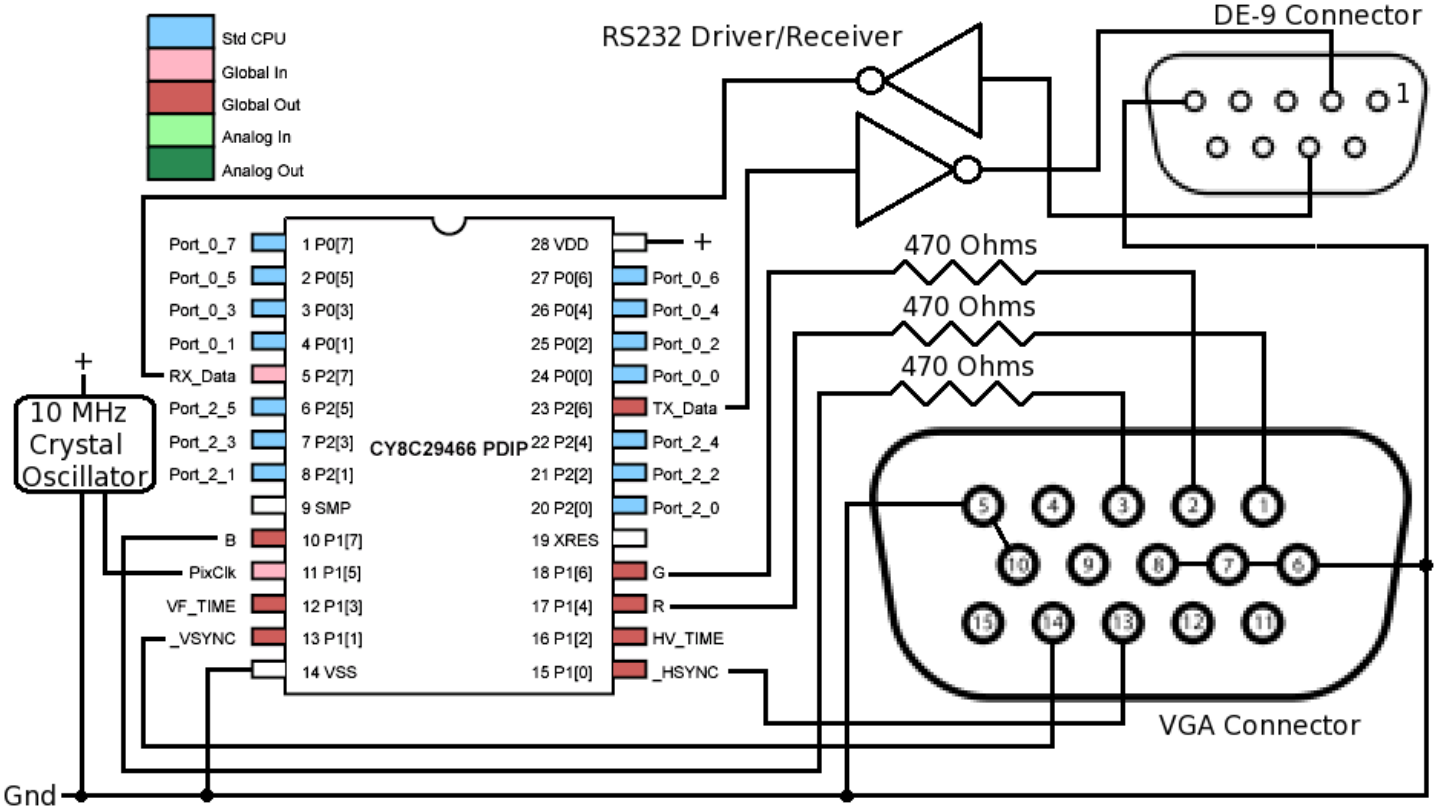

Illustration 8: Elementary Schematic of the PSoC VGA Display Driver.

The only active component, besides the CY8C29466, are the 10 MHz crystal oscillator, the MAX3232 RS232 driver/receiver, and a voltage regulator (the latter two being part of the PSoCEval1 board). The output of the crystal oscillator is connected to line 5 of Port 1 on the PSoC device.

Three 470 Ohm resistors are used to constrain the video channel voltages to range from 0.0 Volts to 0.7 Volts. While these resistor values do not provide a perfect impedance match for the VGA monitor's video channel signals, it is sufficient for the usually short length of a standard VGA monitor cable.

A standard female HD-15 connector is needed to enable connection to a VGA monitor cable. For my prototype, a female VGA connector, taken from an old computer with an integrated VGA display controller, is used. The VGA connector is attached to a short ribbon cable, with a sixteen-pin IDC connector at the other end. Inserting wires into the IDC connector and into the protoboard on the PSoCEval1 board made it easy to prototype the VGA cable interface. A photo of the prototype PSoC VGA Display Driver, as implemented on a PSoCEval1 board, is shown in Illustration 9.

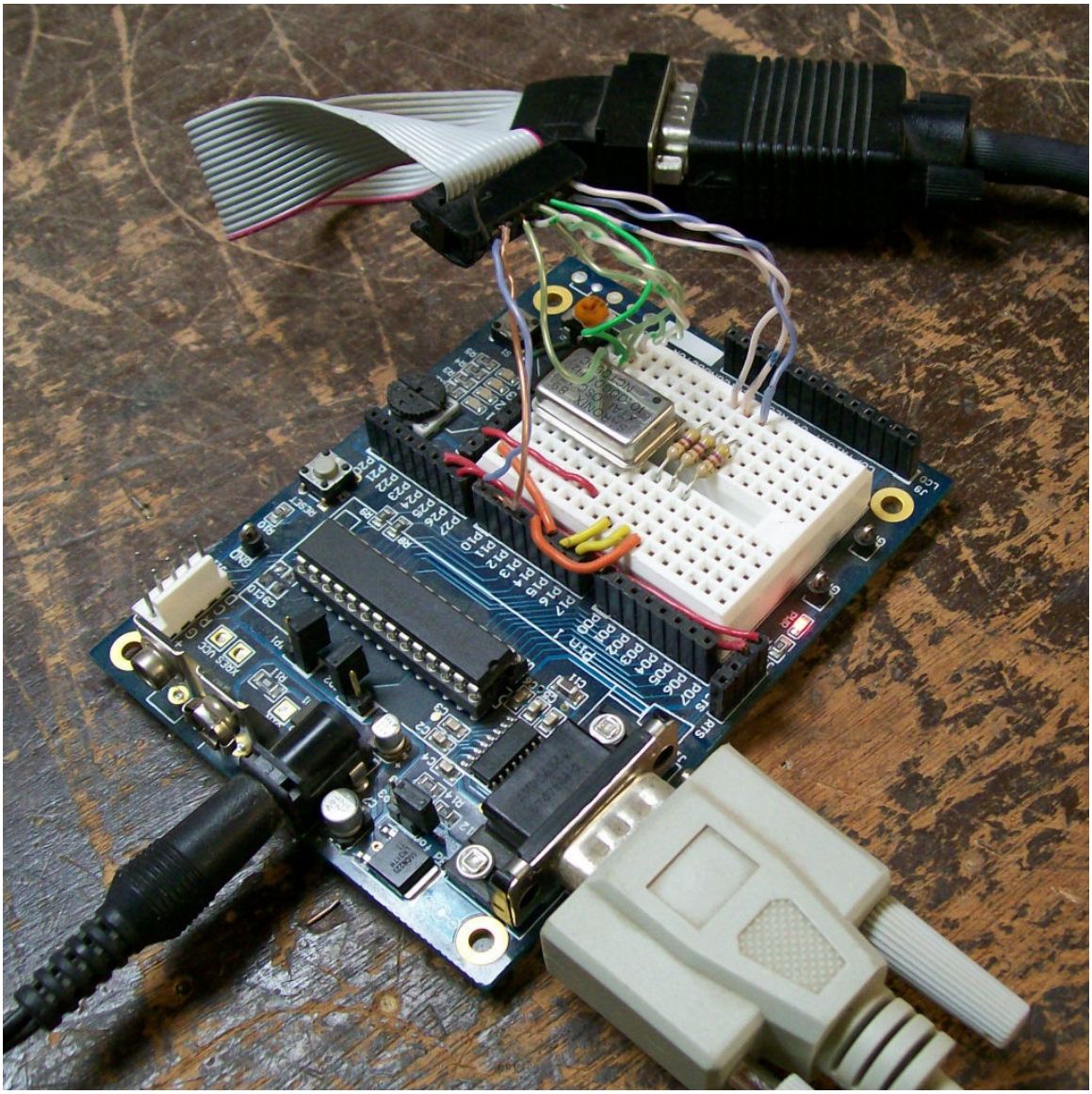

Illustration 9: PSoC VGA Display Driver Prototype on a Cypress PSoCEval1 Board.

Firmware Implementation

The user modules described in the previous sections provide time-critical functions that are necessary for the PSoC VGA Display Driver to work. Firmware, executed by the M8C processor, is still needed to configure, control, and pass data to the user modules. This section provides an overview of the various firmware components. Fine details can be found in the source code contained in the PSoC Designer project for the PSoC VGA Display Driver. The source code is heavily commented.

RAM Allocation

The RAM of the CY8C29466 is divided into eight, 256-byte RAM pages. For the PSoC VGA Display Driver, the pages are divided into three sections. These are listed in Table 2. Page 0 is used for general data storage and the Row Color Scheme array. Page 1 through Page 6 hold the display raster. Page 7 contains the program stack.

Page	Memory Usage	Bytes
0	Function Data Storage and Row Color Scheme Array	256
1	Display Raster	1536
2	Display Raster	
3	Display Raster	
4	Display Raster	
5	Display Raster	
6	Display Raster	
7	Program Stack	256

Table 2: Paged RAM Assignment for the PSoC VGA Display Driver.

The display raster contains 1536 bytes. If each byte represents eight, one-bit pixels, then the display raster contains 12,288 pixels. These can be organized as a 128x96 display raster. This maintains the standard 4:3 aspect ratio of a standard VGA display. Each pixel generated by the PSoC VGA Display Driver can be represented by a 5-by-5 square of VGA pixels.

To allow access to the RAM pages, the code is written to operate in Page Mode 3. In Page Mode 3, direct memory instructions address the RAM page selected by the Current Page Pointer (CUP_PP) and index memory instructions address the RAM page selected by the Index Page Pointer (IDX_PP). The Move Indexed Read Page Pointer (MVR_PP) reserved for use by the interrupt service routine to quickly read bytes from the display raster during the pixel serialization process.

Hardware Initialization

Upon reset, the M8C processor executes the code in *boot.asm*. The initial configuration of the digital modules are configured via the subroutine in *psoccfgtbl.asm*. Upon completion of these basic configurations, program control is passed to the code in *main.asm*. The code in *main.asm* first clears the display raster memory (zero represents

the background color). Next, the color schemes for all 96 rows pixel rows are initialized to a white foreground on a black background. Then, the user modules are started.

The RX_Data interrupt is enabled to allow its interrupt service routine to be called. The RX_Data and TX_Data user modules are then started with no data parity. The HSYNC and VSYNC user modules are then started to produce the horizontal and vertical sync pulses. Next, the LineMask user module is started to generate the video blanking signal. The VGA_Buf1 and VGA_Buf2 user modules are then started to allow the video blanking signal and the serialized pixel stream to reach the color generation function. Note that the PixelData user module is stopped to ensure that no errant data is sent to the VGA display. Finally, all impending interrupts are cleared, GPIO interrupts are enabled, and the general interrupt enable bit is set. Note that the only GPIO interrupt that will occur is on line 3 of Port 1, with the rising edge of the vertical frame time (VF_TIME) signal. This interrupt is set up in the subroutine in *psocconfigtbl.asm*. Whenever the interrupt is invoked, the VGA display is refreshed with the current contents of the display raster. This code will be explained below.

After completion of the initialization, the string "PSoC-VGA V1<CR><LF>!" (excluding quotation marks) is transmitted via repeated calls to the subroutine `_TX_Data_PutChar`. <CR> represents the carriage return character (CR, 0x0d) and <LF> represents the linefeed character (LF, 0x0a). Reception of this string indicates that the PSoC VGA Display Driver is ready for use. Following the transmission of the string, the serial receiver data buffer is reset. At this point the main processing loop is entered.

Command Processing Loop

The main processing loop monitors for the appearance of text characters to display or command messages to execute, as decided by the RX_Data interrupt service routine. The interrupt service routine is the one provided with the RX8 user module with additional code placed within the custom code section. This custom code monitors the incoming serial to determine if either individual text characters are being received for display or if a sequence of characters represents a command that is to be executed.

Upon system reset, the serial receiver is set to character mode. In this mode, received characters are meant for display. As each character is received, the main loop is informed that there is a message in the receive buffer. The main loop detects that the received message is not a command, so the single character is drawn in the display raster. The subroutine `_DrawCharacter` is called to convert the ASCII character to a bitmap that is written into the display raster. Bitmaps for all 256 ASCII characters are stored in the ROM of the CY829466. The ASCII character bit map set is a highly massaged version of the font8x8.h file taken from Kenneth Lemieux's submission to the Circuit Cellar 2004 PSoC High Integration Challenge contest, *Drop Dead Simple Video Generation with the Cypress PSoC*[3]

The character's bitmap is placed at the current location specified by the character cursor. The character cursor specifies a location on a twelve row by sixteen column character grid superimposed on the display raster. The cursor location is advanced by one character, wrapping to the next line or back to the top of the display raster as necessary. This is performed by a call to the subroutine `_AdvanceCursor`. Each character received will also be echoed back serially through the TX_Data user module

3 http://www.circuitcellar.com/PSOC2004/winners/DE/c2959.htm

by calling the subroutine `_TX_Data_PutChar`.

If an escape character (ESC, 0x1b) is received, the serial receiver software switches into command mode. Subsequent characters are gathered into the receive buffer until a carriage return character (CR, 0x0d) is received. When the main loop is notified that a message has been received, it detects that the message is a command. The characters in the receive buffer are scanned by the main loop to determine if a properly formated command has been received. The subroutine `_RX_Data_szGetParam` is called to find the location of each parameter in the command string. The subroutines `_CommandMatch` and `_ScanDecimalNumber` are called to validate the parameters. If the command is rejected, a question mark character (?, 0x3F) is transmitted by the TX_Data user module. If the command is accepted, the command is processed and an exclamation point character (!, 0x21) is transmitted. In either case, the main loop sets the receiver software state back to character mode and resets the serial receiver data buffer.

Nine commands are available for graphics operations, color selection, and cursor manipulation. All commands are preceded by the escape character (ESC, 0x1b) and followed by the carriage return character (CR, 0x0d).

<ESC>B<CR> Subsequent drawing operations are performed using the background color. Characters are drawn with the background color for the font on a backdrop of the foreground color. The subroutine `_SetBackgroundColor` implements this command.

<ESC>C r s<CR> Starting at the current pixel row position, set each row, to and including row "r", to the color scheme "s". The current pixel location will be set to row “r”. The column for the current pixel location will not be changed. Note that there are 96 row positions (0 - 95) and 64 color schemes (0 – 63). The foreground/background color combination for each color scheme is specified in Table 3. The eight color scheme numbers marked with an asterisk indicate that the display raster will not be visible. Only the indicated color will be displayed. This can be useful for blanking the screen without deleting the current content of the display raster. The subroutine `_SetColorSchemeToRow` implements this command.

<ESC>F<CR> Subsequent drawing operations are performed using the foreground color. Characters are drawn with the foreground color for the font on a backdrop of the background color. The subroutine `_SetForegroundColor` implements this command.

<ESC>L c r<CR> Draw a line from the current pixel location to the pixel at column "c" and row "r". The current pixel will now be at that location. Note that there are 128 column positions (0 - 127) and 96 row positions (0 – 95). The subroutine `_DrawLineToPixel`, which uses the Bresenham line algorithm, implements this command.

<ESC>M c r<CR> Set the current pixel cursor position to the pixel at column "c" and row "r". Note that there are 128 column positions (0 - 127) and 96 row positions (0 – 95). The subroutine `_MoveToPixel` implements this command.

<ESC>P c r<CR> Draw a pixel at column "c" and row "r". The current pixel location will not be changed. Note that there are 128 column positions (0 - 127) and 96 row positions (0 – 95). The subroutine `_DrawPixel` implements this command.

Scheme Number	Foreground Color	Background Color	Scheme Number	Foreground Color	Background Color
0*	Black	Black	32	Black	Blue
1*	Red	Red	33	Red	Magenta
2	Black	Red	34	Black	Magenta
3	Red	Black	35	Red	Blue
4*	Green	Green	36	Green	Cyan
5*	Yellow	Yellow	37	Yellow	White
6	Green	Yellow	38	Green	White
7	Yellow	Green	39	Yellow	Cyan
8	Black	Green	40	Black	Cyan
9	Red	Yellow	41	Red	White
10	Black	Yellow	42	Black	White
11	Red	Green	43	Red	Cyan
12	Green	Black	44	Green	Blue
13	Yellow	Red	45	Yellow	Magenta
14	Green	Red	46	Green	Magenta
15	Yellow	Black	47	Yellow	Blue
16*	Blue	Blue	48	Blue	Black
17*	Magenta	Magenta	49	Magenta	Red
18	Blue	Magenta	50	Blue	Red
19	Magenta	Blue	51	Magenta	Black
20*	Cyan	Cyan	52	Cyan	Green
21*	White	White	53	White	Yellow
22	Cyan	White	54	Cyan	Yellow
23	White	Cyan	55	White	Green
24	Blue	Cyan	56	Blue	Green
25	Magenta	White	57	Magenta	Yellow
26	Blue	White	58	Blue	Yellow
27	Magenta	Cyan	59	Magenta	Green
28	Cyan	Blue	60	Cyan	Black
29	White	Magenta	61	White	Red
30	Cyan	Magenta	62	Cyan	Red
31	White	Blue	63	White	Black

*Table 3: Foreground/Background Color Schemes. *Scheme hides the display raster.*

<ESC>R c r<CR> Draw a rectangle from the current pixel cursor position to the pixel at column "c" and row "r". The current pixel will now be at that location. Note that there are 128 column positions (0 - 127) and 96 row positions (0 - 95). The subroutine `_DrawFilledRectangle` implements this command.

<ESC>T c r<CR> Set the current text cursor position to the character at column "c" and row "r". Note that there are sixteen column positions (0 - 15) and twelve row positions (0 - 11). The subroutine `_MoveCursorTo` implements this command.

<ESC>V<CR> Activate the video display. The subsequent reception of any ASCII character by the RX_Data interrupt service routine will result in the deactivation the video display. This is done allow for the uninterrupted reception of additional characters. This command re-enables GPIO interrupts.

The subroutines invoked by the commands can be found in the file *draw_utils.asm*.

VGA Screen Refresh

Approximately every 1/60 of a second, the rising edge of the VF_TIME signal triggers a GPIO interrupt. The interrupt service routine, found in the file *psocgpioint.asm*, is executed. Being an interrupt, the Page Mode bits are cleared automatically, which forces RAM addressing to Page Mode 0. In order to access all of the RAM of the CY829466, the custom code in the routine executes the `RAM_SET_NATIVE_PAGING` macro, which sets the Page Mode bits back to Page Mode 2. After the required saving of the accumulator, the index register, and the Page addressing registers to the stack, the subroutine `_DisplayRasterVGA` is called. After returning from the subroutine call, the registers and accumulator are recalled and processing is returned to the main processing loop.

The subroutine `_DisplayRasterVGA`, found in the file *vga.asm*, performs a significant amount of processing. In fact, when screen refreshing is enabled, more processing time is spent in the execution of this subroutine than in the main processing loop. This is due to the time-critical requirement to supply data from the display raster for pixel serialization by the PixelData user module. In addition, the Color Generation function is adjusted for every pixel line to display the color scheme for that line, as contained in the Row Color Scheme array (`row_color`).

At the beginning of the subroutine, various memory values are initialized for proper counting of the number of pixel lines displayed in the frame (`line_count`), the byte counter for RAM page increments (`byte_count`), the current RAM page number for the display raster (`page_number`), and the pointer to the display raster (`pbyte_ptr`). The Index Page Pointer (IDX_PP) is set to Page 0 so the Row Color Scheme array can be addressed. Also, the horizontal video time line (`H_VideoTime_LINE`) signal is enabled by changing its LUT from a low signal output to an A AND B logic function.

At the start of each pixel line, the Move Indexed Read Page Pointer (MVR_PP) is set to the RAM page that contains the raster data for that pixel line. The video lines per pixel line (`repeat_count`) is initialized to the value of the constant `repeat_lines` (five video lines per pixel line). Lastly, the Row Color Scheme array is accessed to obtain two bytes associated with that pixel line to set up the LUTs of the Color Generation function.

While an interrupt is used to invoke the subroutine `_DisplayRasterVGA` at the appropriate frame time, timing within video lines is performed through polling of

register bits. The M8C microprocessor's interrupt latency is too slow to support the critical events that occur within a video line. Detection of the falling edge of the horizontal sync pulse (_HSYNC) is accomplished with the `WAIT_FOR_LOW` macro, which tests for a zeroed bit on Port 1, as specified by the constant `_HSYNC_LINE`.

Once the falling edge of _HSYNC occurs, the subroutine prepares for the display of pixel bytes for that line. Note that at this point in time, the PixelData user module is not being clocked. The first pixel byte for the current line is fetched from the display raster. The PixelData user module is stopped to force it back to an idle state. Then it is restarted and the byte of pixel data is written into PixelData's SPIM transmit buffer. Since the PixelData user module is not being clocked, the byte is not loaded into the transmit shift register. When the video line time (HV_TIME) signal goes high, the clock starts and the SPIM loads the pixel data into the transmit shift register for serialization.

The code in the macro `PushPixelByte` is executed fifteen times to serialize the remaining fifteen pixel bytes. In the macro, the code first reads another pixel byte from the display raster. Then the SPIM status register is polled to determine if the previous pixel byte has been moved from the SPIM transmit buffer to the SPIM transmit shift register. Once this has occurred, the pixel byte that was read is written into the SPIM transmit buffer.

After the video line has been serialized, the subroutine code checks to see if there are any pending interrupts. The only other interrupt that could occur would be caused by the reception of a byte by the RX_Data user module. If this occurs, the `_DisplayRasterVGA` subroutine is exited early and the video frame interrupt is terminated. This is done to ensure that no serial data is missed. Note that no further video frame interrupts will be allowed to occur until all serial data reception and processing is completed and the <ESC>V<CR> command is executed.

If there are no pending interrupts, memory values are decremented and checked to determine if all of the video lines for the pixel line have been completed. If not, another video line is drawn using the same pixel bytes for the pixel line. If all of the video lines for the pixel line are completed, then the display of the next pixel line will be performed. If all of the pixel lines have been displayed, then the video frame is completed. The horizontal video time line (`H_VideoTime_LINE`) signal is disabled by changing its LUT from an A AND B logic function to a low signal output. The PixelData user module is also stopped to ensure that the SPIM function is reset.

A Demonstration

I've have provided a small demo for displaying text and graphics on the PSoC VGA Display Driver. This demo exercises all of of the commands described previously. The listing of the demo is provided in Table 4. Note that the listing is split into two columns, with the line numbers indicating the order of transmission to the PSoC VGA Display Driver. The ^[character sequence represents the escape character (<ESC>, 0x1b) that precedes a command. Each string has a implied carriage return character (<CR>, 0x0d) at its end. Note that carriage return characters in non-command strings are ignored at reception. The comments are provided for understanding only. They are not part of the data transmitted to the PSoC VGA Display Driver. A photograph of a VGA screen displaying the demo is shown in Illustration 9

Line	String	Comment	Line	String	Comment
1	^[M 0 0	Move pixel cursor to column 0, row 0	44	^[M 0 24	Move pixel cursor to column 0, row 24
2	^[C 16 41	Red/White color scheme through row 16	45	^[L 64 16	Draw a line to column 64, row 16
3	^[C 34 10	Black/Yellow color scheme through row 34	46	^[L 127 24	Draw a line to column 127, row 24
4	^[C 49 56	Blue/Green color scheme through row 49	47	^[L 64 33	Draw a line to column 64, row 33
5	^[C 78 27	Magenta/Cyan color scheme through row 78	48	^[L 0 24	Draw a line to column 0, row 24
6	^[C 79 0	Black/Black color scheme through row 79	49	^[M 48 21	Move pixel cursor to column 48, row 21
7	^[C 95 10	Black/Yellow color scheme through row 95	50	^[R 80 28	Draw rectangle to column 80, row 28
8	^[M 64 0	Move pixel cursor to column 64, row 0	51	^[B	Draw with the background color
9	^[R 127 15	Draw rectangle to column 127, row 14	52	^[M 50 23	Move pixel cursor to column 50, row 23
10	^[M 40 34	Move pixel cursor to column 40, row 34	53	^[R 78 26	Draw rectangle to column 78, row 26
11	^[R 127 48	Draw rectangle to column 127, row 48	54	^[F	Draw with the foreground color
12	^[T 3 0	Move text cursor to column 3, row 0	55	^[M 0 77	Move pixel cursor to column 0, row 77
13	^[F	Draw with the foreground color	56	^[L 127 49	Draw a line to column 127, row 49
14	PSoC	Draw the text string “PSoC”	57	^[M 0 77	Move pixel cursor to column 0, row 77
15	^[T 9 0	Move text cursor to column 9, row 0	58	^[L 127 52	Draw a line to column 127, row 52
16	^[B	Draw with the background color	59	^[M 0 77	Move pixel cursor to column 0, row 77
17	VGA	Draw the text string “VGA”	60	^[L 127 55	Draw a line to column 127, row 55
18	^[T 0 1	Move text cursor to column 0, row 1	61	^[M 0 77	Move pixel cursor to column 0, row 77
19	^[F	Draw with the foreground color	62	^[L 127 58	Draw a line to column 127, row 58
20	Display	Draw the text string “Display”	63	^[M 0 77	Move pixel cursor to column 0, row 77
21	^[T 9 1	Move text cursor to column 9, row 1	64	^[L 127 61	Draw a line to column 127, row 61
22	^[B	Draw with the background color	65	^[M 0 77	Move pixel cursor to column 0, row 77
23	Driver	Draw the text string “Driver”	66	^[L 127 64	Draw a line to column 127, row 64
24	^[T 4 10	Move text cursor to column 4, row 10	67	^[M 0 77	Move pixel cursor to column 0, row 77
25	^[F	Draw with the foreground color	68	^[L 127 67	Draw a line to column 127, row 67
26	^[T 1 5	Move text cursor to column 1, row 5	69	^[M 0 77	Move pixel cursor to column 0, row 77
27	PSoC	Draw the text string “PSoC”	70	^[L 127 70	Draw a line to column 127, row 70
28	^[B	Draw with the background color	71	^[M 0 77	Move pixel cursor to column 0, row 77
29	Developer	Draw the text string “Developer”	72	^[L 127 73	Draw a line to column 127, row 73
30	^[F	Draw with the foreground color	73	^[M 0 77	Move pixel cursor to column 0, row 77
31	^[T 4 10	Move text cursor to column 4, row 10	74	^[L 127 76	Draw a line to column 127, row 76
32	(C) 2008	Draw the text string “(C) 2008”	75	^[M 1 50	Move pixel cursor to column 1, row 50
33	^[T 2 11	Move text cursor to column 2, row 11	76	^[R 48 64	Draw rectangle to column 48, row 64
34	Mac A. Cody	Draw the text string “Mac A. Cody”	77	^[B	Draw with the background color
35	^[M 25 36	Move pixel cursor to column 25, row 36	78	^[M 3 52	Move pixel cursor to column 3, row 52
36	^[R 27 38	Draw rectangle to column 27, row 38	79	^[R 46 62	Draw rectangle to column 46, row 62
37	^[P 26 34	Draw a pixel at column 26, row 34	80	^[F	Draw with the foreground color
38	^[P 23 37	Draw a pixel at column 23, row 37	81	^[M 5 54	Move pixel cursor to column 5, row 54
39	^[P 29 37	Draw a pixel at column 29, row 37	82	^[R 44 60	Draw rectangle to column 44, row 60
40	^[P 26 40	Draw a pixel at column 26, row 40	83	^[B	Draw with the background color
41	^[B	Draw with the background color	84	^[M 7 56	Move pixel cursor to column 7, row 56
42	^[P 26 37	Draw a pixel at column 26, row 37	85	^[R 42 58	Draw rectangle to column 42, row 58
43	^[F	Draw with the foreground color	86	^[V	Activate the video display

Table 4: Listing of the Data Used for the Display Driver Demo .

Illustration 10: PSoC VGA Display Driver Demo.

Conclusion

The PSoC VGA Display Driver provides the ability to use inexpensive VGA displays in embedded applications where low-resolution color text and graphics are needed. The VGA sync signal generation and pixel display techniques that have been developed can be used as the basis of other video applications. PSoC digital blocks provide the critical timing and data serialization functions that make this application possible. On the Cypress CY8C29466 PSoC, additional digital blocks and all analog blocks are available for other applications. If black and white text and graphics are sufficient for an application, the Color Generation function can be removed, freeing additional digital blocks and memory resources.

I want to thank the members of the PSoCDeveloper[4] forum for their helpful suggestions and solutions to problems that I have had during the development of the PSoC VGA Display Driver. They have proven to be a responsive and friendly group of individuals. Collectively, they are providing a great service to the PSoC community and are doing a lot to popularize the PSoC as a very flexible embedded solution.

4 http://www.psocdeveloper.com