Gammon Forum

See www.mushclient.com/spam for dealing with forum spam. Please read the MUSHclient FAQ!

Arduino Uno output to VGA monitor

Postings by administrators only.

Refresh page

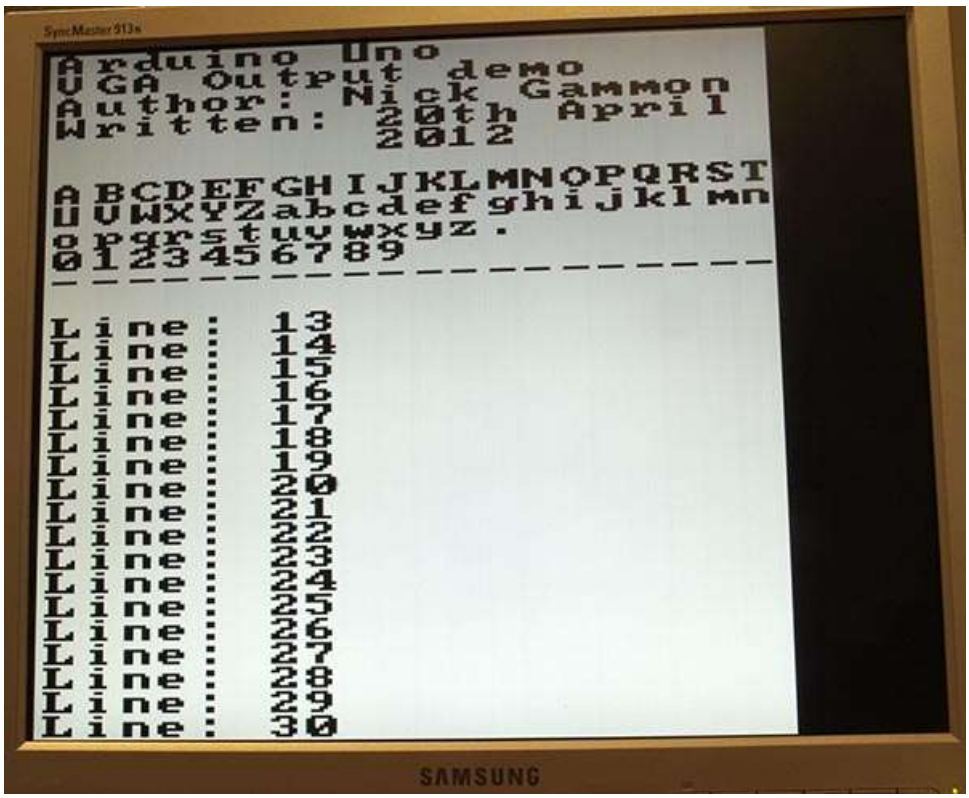
Posted by Nick Gammon Australia (22,775 posts) bio *Forum Administrator*

Date Sat 21 Apr 2012 04:12 AM (UTC)

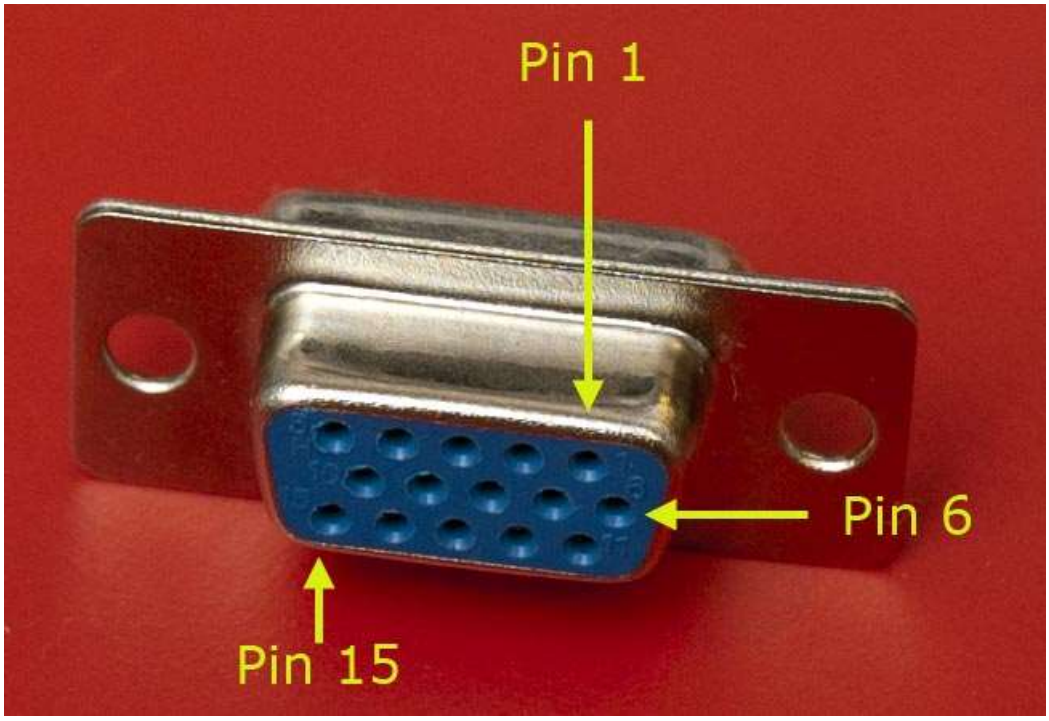
Amended on Wed 24 Feb 2016 07:54 PM (UTC) by Nick Gammon

Message

This post shows how you can output VGA signals to a monitor, turning your Uno (or similar) processor into a "video card". Example:



The only extra hardware used here was 5 resistors, and a DB15 socket to plug the monitor cable into:



Watch live demo:

VGA Output on Arduino with text

Nick Gammon

00:27

VGA Timing

To understand how it all works, we need to look at how VGA signal timing works.

There are 5 signals being sent to the monitor:

- Vertical Synchronization (sync) (TTL)

- Horizontal Synchronization (sync) (TTL)
- Red analog data (0 to 0.7V)
- Green analog data (0 to 0.7V)
- Blue analog data (0 to 0.7V)

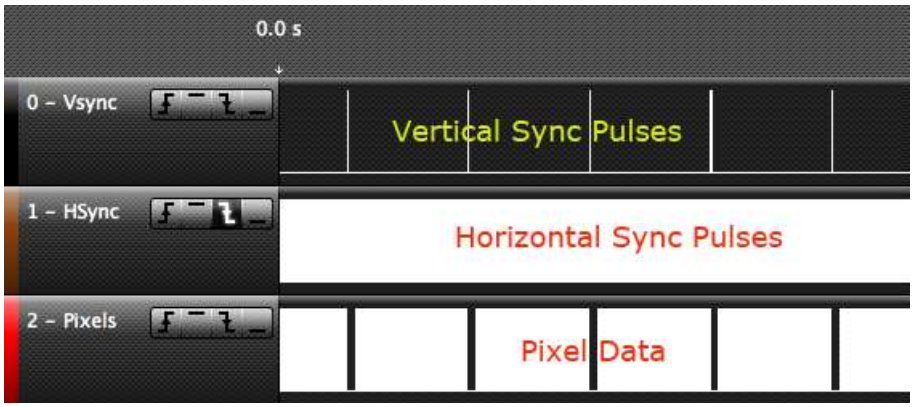
For timing reasons we don't have time to output separate red, green, and blue levels, so they are physically connected at the socket, giving white.

[EDIT] However see follow-up post showing how you **can** do colour.

Thus there are three important signals now:

- Vertical sync pulses
- Horizontal sync pulses
- Pixel data

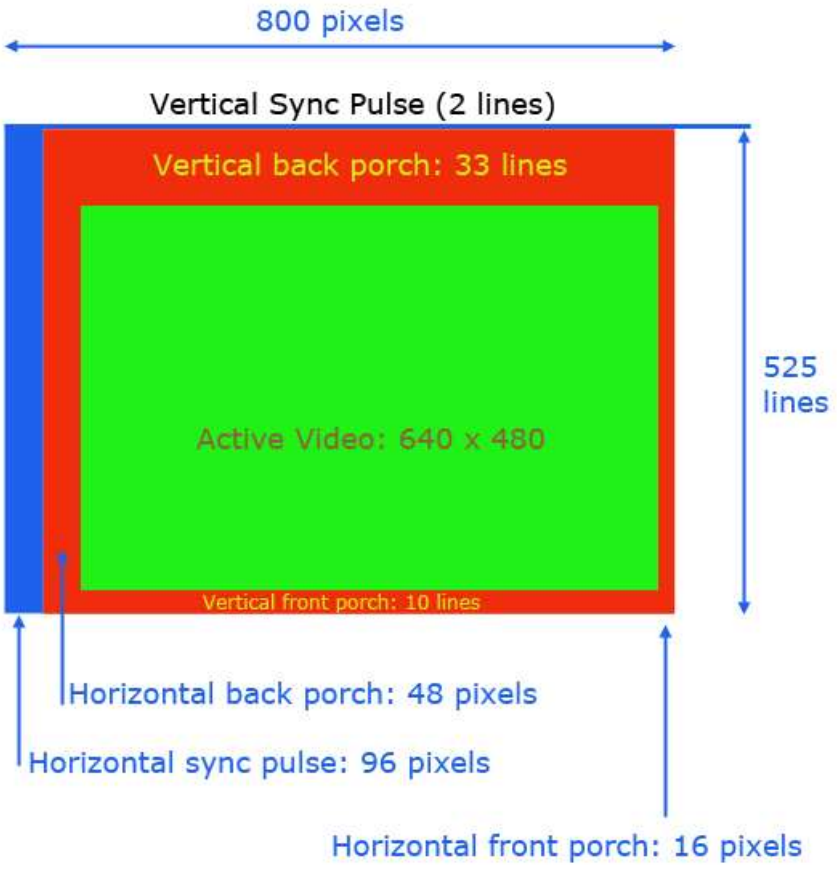
On the logic analyzer we can see an overall view:



The two lower signals are just blocks of white right now - we'll have to zoom in to see them better.

Overall structure

This diagram shows the various timings involved:



Assuming for discussion that we have a 640 x 480 resolution screen, the hardware needs a little extra time to move the electron beam (in the days when we used CRT monitors) from one side of the screen to the other, and from the bottom back to the top. These are the front and back "porches" (don't ask me why the name).

The screen refresh starts off with a **vertical** sync pulse, which tells the monitor to reset to the top of the screen. It has the "back porch" number of lines to get ready to start drawing, and to draw a blank area at the top. Then it draws the image, and has the "front porch" extra lines to draw the blank information at the bottom of the screen.

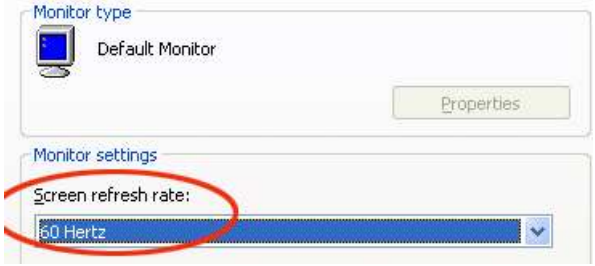
Meanwhile, for each line, there is a **horizontal** sync pulse which signals the start of that line, followed by another delay to give the beam time to get ready, then it draws the line, and has some extra "front porch" time over for the blank part at the end of the line.

Vertical Sync

Let's start with the vertical sync pulses. In fact, we'll show how all the timing data can be derived from three figures:

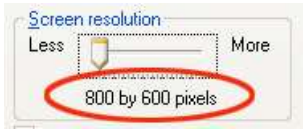
- The screen refresh rate (eg. 60 Hz)
- The screen resolution (eg. 640 x 480)

So what is the refresh rate? It's the rate at which **the entire screen is redrawn**. If you ever looked at Windows screen resolution you probably saw something like this:



That's the refresh rate: 60 Hz (60 times a second). This figure was probably originally chosen because it is the mains frequency in the USA, so that would minimize the artifact of mains hum bars appearing on the screen, in the days of CRT monitors.

The other figure of interest is the screen resolution, for example:



In our case we are going for the minimum we can (640 x 480) and seeing where those figures lead us.

Starting with the refresh rate, a 60 Hz refresh rate will require a "vertical sync" pulse 60 times a second, or a period of 1/60 (16.66 ms).

This screenshot shows that the vertical sync pulses are indeed (nearly) 16.66 ms apart:

Observed figures might differ from theory by about 1% due to the internal clock of the processor not being 100% accurate.

Generating vertical sync pulses

This code, in setup, generates the vertical sync pulses:

```
// Timer 1 - vertical sync pulses
pinMode (vSyncPin, OUTPUT);
Timer1::setMode (15, Timer1::PRESCALE_1024, Timer1::CLEAR_B_ON_COMPARE);
OCR1A = 259;  // 16666 / 64 µs = 260 (less one)
OCR1B = 0;    // 64 / 64 µs = 1 (less one)
```

The clock period is 62.5 ns (1/16000000). By applying a prescaler of 1024 the timer counts up once every 64 µs. So to get a period of 1/60 of a second (16666 µs) we need to count up to 260 (16666 / 64 = 260). Since the timer

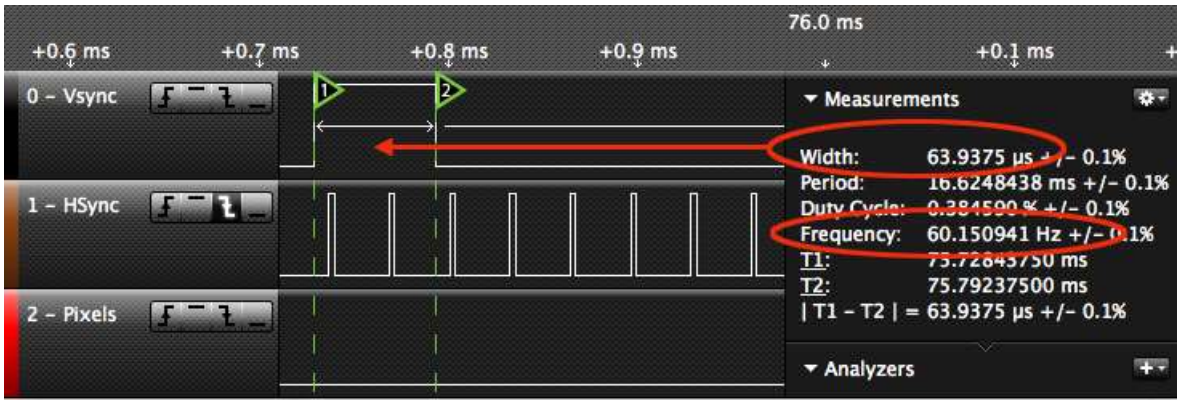
counts are zero-relative we set OCR1A to count up to 259. This sets the frequency of Timer 1.

Then we need to set the pulse width by putting the correct value in OCR1B. We want a pulse width of two lines. One line is 1 / 60 / 525, namely 31.7 µs. So two lines would be 63.4 µs. This is close enough to exactly one timer count (64 µs). So OCR1B is set to zero (being zero-relative, making it zero gives a count of 1).

Timer 1 is configured to "clear B on compare" which effectively means that it toggles the output pin (D10 on the Uno) so it is high for the duty cycle width (64 µs) and low the rest of the time. If you wanted the opposite sync pulse polarity, change CLEAR_B_ON_COMPARE to SET_B_ON_COMPARE.

We can see from the logic analyzer output that the vertical sync pulses are indeed happening at 60 Hz, and that the pulse width is 63.9 µs.

So far so good ...



The timer is also set up to generate an interrupt, which is used to tell the code that we are starting another vertical cycle, by setting the line count to zero:

```
ISR (TIMER1_OVF_vect)
  {
  vLine = 0;
  messageLine = 0;
  backPorchLinesToGo = verticalBackPorchLines;
  } // end of TIMER1_OVF_vect
```

Horizontal sync

Horizontal sync pulses tell the monitor when to start drawing each line.

To calculate the horizontal sync frequency we need to divide the overall frame rate (60 Hz) by the number of total lines (525 if you count the sync pulse itself, and the front and back porches).

In other words:

```
(1/60) / 525 * 1e6 = 31.74 µs
```

We also need to know the sync pulse width. That is documented to be 96 pixels, so we need to know the width of one pixel. That would be the figure above, divided by 800 (being the total screen width including the sync pulse,

and front and back porches).

Thus that is:

```
((1/60) / 525 * 1e9) / 800 = 39.68  ns
1 / (((1/60) / 525 * 1e6) / 800) = 25.2 MHz
```

So our pixel width is 39.68 ns and the pixel clock is 25.2 mHz.

The horizontal sync pulse width is 96 pixels, so we want a pulse of:

```
96 * 39.68 ns = 3.8 µs
```

Generating horizontal sync pulses

Now we are ready to set up the timer for the horizontal sync pulses:

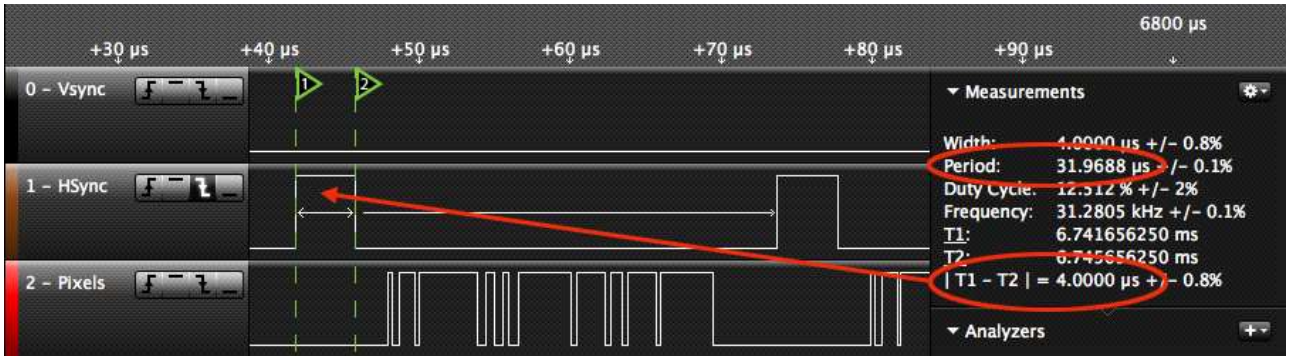
```
  // Timer 2 - horizontal sync pulses
  pinMode (hSyncPin, OUTPUT);
  Timer2::setMode (7, Timer2::PRESCALE_8, Timer2::CLEAR_B_ON_COMPARE);
  OCR2A = 63;   // 32 / 0.5 µs = 64 (less one)
  OCR2B = 7;    // 4 / 0.5 µs = 8 (less one)
```

The clock period is 62.5 ns (1/16000000). By applying a prescaler of 8 the timer counts up once every 0.5 µs. So to get a period of 32 µs we need to count up to 64 (32 / 0.5 = 64). Since the timer counts are zero-relative we set OCR2A to count up to 63. This sets the frequency of Timer 2.

Then we need to set the pulse width by putting the correct value in OCR2B. We want a pulse width of 96 pixels (3.8 µs). So OCR2B is set to 7 (being zero-relative, making it 7 gives a count of 8, which is 4 / 0.5).

Again, if you wanted the opposite sync pulse polarity, change CLEAR_B_ON_COMPARE to SET_B_ON_COMPARE.

We can see from the logic analyzer output that the horizontal sync pulses are indeed happening every 31.9 µs, and that the pulse width is 4 µs.



The timer is also set up to generate an interrupt, which has the sole purpose of waking the processor up from sleep, so it can draw each line with exactly the same delay after the pulse. If it wasn't asleep, there would be a variation of two to three clock cycles (since an interrupt cannot occur during a single instruction) and this gives very bad-looking "jitter" on the screen.

```
ISR (TIMER2_OVF_vect)
  {
  } // end of TIMER2_OVF_vect
```

Pixel data

Now things get tricky ...

Let's see how long we have to draw 640 pixels:

```
((1/60) / 525 * 1e9) / 800  * 640 = 25396.82 ns (25.39 µs)
```

It just can't be done on this processor. The clock period itself is only 62.5 ns, so there is no way we can output a pixel every 39.68 ns (that is: 25396.82 / 640 = 39.68).

The fastest way we can get bits "out the door" is using the SPI hardware. That can run at a maximum clock rate of twice the system clock, that is, one pixel every 125 ns. So we will have to settle for having the pixels 4 times as wide. A horizontal resolution therefore of 160 pixels. Since we will display 8-pixel characters, that gives us 20 characters per line.

We also don't have enough memory to even hold those pixels all at once. Even at a bit per pixel, that would be:

```
160 * 480 / 8 = 9600 bytes
```

Since the processor only has 2048 bytes of memory there is no way we can hold any sort of bitmap. So, we will have to decode text into pixels "on the fly" for every line. This itself takes time. Especially since to hold 256 characters as 8 bytes each (8 x 8 pixels) takes 2048 bytes. So we can't hold the font data in RAM either.

So, the fonts have to go into PROGMEM (program memory) and be accessed on the fly for every line, using "program memory read".

I am using the USART in SPI mode. This is double-buffered so hopefully should give us a slight edge in performance. Even doing that isn't perfect because of what I have read of as the "9th bit problem". That is, the USART sends out one additional "high" bit per byte. This looks very annoying if you are doing white-on-black because there is a white bar between each letter. To hide this, we invert the fonts to give black on white text. Then the extra bit just looks like a space between the letters.

The line drawing is done in doOneScanLine which is called directly after being woken from sleep. As mentioned above, the sleep is necessary to make sure that we start each scan line at exactly the same time after the sync pulse:

```
void loop()
  {
  // sleep to ensure we start up in a predictable way
  sleep_mode ();
  doOneScanLine ();
 }  // end of loop
```

Inside doOneScanLine we check if we have done the back porch lines, and if not, exit. If it is time to draw something, we pre-load some pointers (to save time) and then execute this loop:

```
  // blit pixel data to screen
  while (i--)
    UDR0 = pgm_read_byte (linePtr + (* messagePtr++));
```

For each byte in the current "message" (text) line, we look up the relevant pixel (font) data in program memory, and stuff that into the USART "send" register.

The font data has been deliberately structured to save a multiply by 8. Initially I had each character together (so from one letter to the next was 8 bytes, since it takes 8 bytes per character). Unfortunately, multiplying by 8 (even if you do it via a shift left) just took too long. So instead, the font table was restructured so all the data per scan line was together, not all the data for one letter. Also the font is pre-inverted (to white on black) to save doing that at drawing time.

There aren't any delays or tests for "is SPI ready?" in that loop. Let's see why not. The code for that loop is:

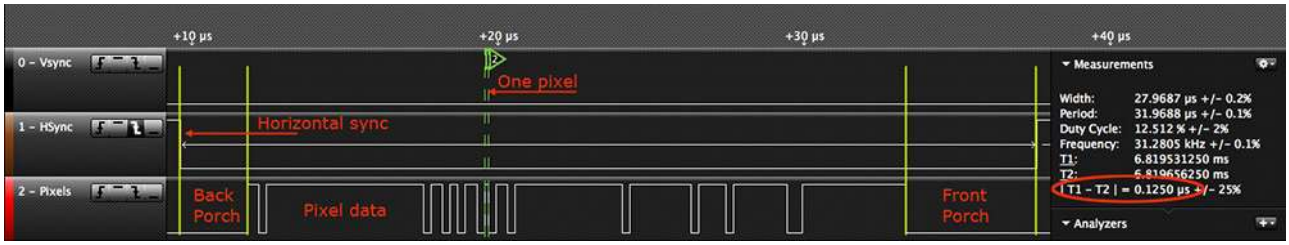
```
  // blit pixel data to screen
  while (i--)
    UDR0 = pgm_read_byte (linePtr + (* messagePtr++));
 (1)    f04:    32 2f           mov     r19, r18
 (1)    f06:    22 27           eor     r18, r18
 (2)    f08:    0b c0           rjmp    .+22            ; 0xf20

----
 (2)    f0a:    ed 91           ld      r30, X+
 (1)    f0c:    ff 27           eor     r31, r31
 (1)    f0e:    e7 fd           sbrc    r30, 7
 (1)    f10:    f0 95           com     r31
 (1)    f12:    e2 0f           add     r30, r18
 (1)    f14:    f3 1f           adc     r31, r19
 (1)    f16:    e8 59           subi    r30, 0x98       ; 152
 (1)    f18:    ff 4f           sbci    r31, 0xFF       ; 255
 (3)    f1a:    e4 91           lpm     r30, Z+
```

```
 (2)    f1c:    e0 93 c6 00     sts     0x00C6, r30
 (1)    f20:    81 50           subi    r24, 0x01       ; 1
 (2)    f22:    98 f7           brcc    .-26            ; 0xf0a

----
17 cycles in loop = 1062.5 ns
```

Apart from the initial setup at the start of the loop (done once) there are 17 cycles in the loop. I added the cycle count for each instruction at the start of the line.

Now it takes 16 cycles to output 8 bits (since each bit takes 125 ns) so each byte is transferred in 16 cycles, and the loop above takes 17. So that gives us a 1-cycle gap between letters, which apparently we get anyway because of the way that SPI works. So the loop is exactly the right length.

One scan line being drawn:



Note that the thinnest pixel (arrowed) is 125 ns long. That's the shortest we can do.

Finished code

The finished code is remarkably short. This is what it looks like without any ability to accept incoming text changes:

```
/*
 VGA video generation

 Author:   Nick Gammon
 Date:     20th April 2012

 Example code without I2C communications.

 Connections:

 D1 : Pixel output (180 ohms in series) (connect to R, G, B tied together)   -->
 D3 : Horizontal Sync (68 ohms in series) --> Pin 13 on DB15 socket
 D10 : Vertical Sync (68 ohms in series) --> Pin 14 on DB15 socket

 Gnd : --> Pins 5, 6, 7, 8, 10 on DB15 socket

*/

#include <TimerHelpers.h>
#include <avr/pgmspace.h>
#include "screenFont.h"
#include <avr/sleep.h>

#define BETA_ARDUINO ARDUINO < 100

const byte pixelPin = 1;     // <------- Pixel data
const byte hSyncPin = 3;     // <------- HSYNC
const byte MSPIM_SCK = 4;    // <-- we aren't using it directly
const byte vSyncPin = 10;    // <------- VSYNC

const int horizontalBytes = 20;  // 160 pixels wide
const int verticalPixels = 480;  // 480 pixels high
```

```
const byte i2cAddress = 42;

// Timer 1 - Vertical sync

// output    OC1B   pin 16  (D10) <------- VSYNC

//   Period: 16.64 ms (60 Hz)
//      1/60 * 1e6 = 16666.66 µs
//   Pulse for 64 µs  (2 x HSync width of 32 µs)
//    Sync pulse: 2 lines
//    Back porch: 33 lines
//    Active video: 480 lines
//    Front porch: 10 lines
//       Total: 525 lines

// Timer 2 - Horizontal sync

// output    OC2B   pin 5  (D3)   <------- HSYNC

//   Period: 32 µs (31.25 kHz)
//      (1/60) / 525 * 1e6 = 31.74 µs
//   Pulse for 4 µs (96 times 39.68 ns)
//    Sync pulse: 96 pixels
//    Back porch: 48 pixels
//    Active video: 640 pixels
//    Front porch: 16 pixels
//       Total: 800 pixels

// Pixel time =  ((1/60) / 525 * 1e9) / 800 = 39.68  ns
```

This is the file "screenFont.h" which has the font patterns in it:

```
// bit patterns for the CP437 font, ROTATED, INVERTED

// To save rendering time, they are ordered by line (ie. line 1, line 2) and the
//  indexed by character.

byte screen_font [8] [256] PROGMEM = {
// Row 1
 {
  0xFF, 0xC0, 0xC0, 0xC9, 0xF7, 0xE3, 0xF7, 0xFF, 0x80, 0xFF, 0x80, 0xF8, 0xE1,
  0xBF, 0xFE, 0xF3, 0xCC, 0xC0, 0xE0, 0xFF, 0xF3, 0xF3, 0xF3, 0xFF, 0xFF, 0xFF,
  0xFF, 0xE7, 0xC9, 0xC9, 0xE7, 0xFF, 0xE3, 0xCF, 0xF3, 0xCF, 0xFF, 0xFF, 0xFF,
  0xC1, 0xE7, 0xC3, 0xC3, 0xF1, 0x81, 0xE3, 0x81, 0xC3, 0xC3, 0xFF, 0xFF, 0xF3,
  0xC1, 0xE7, 0x81, 0xE1, 0x83, 0x80, 0x80, 0xE1, 0x99, 0xC3, 0xF0, 0x8C, 0x87,
  0x81, 0xC3, 0x81, 0xC3, 0x81, 0x99, 0x99, 0x9C, 0x9C, 0x99, 0x80, 0xC3, 0x9F,
  0xE7, 0xFF, 0x8F, 0xFF, 0xF1, 0xFF, 0xE3, 0xFF, 0x8F, 0xE7, 0xF9, 0x8F, 0xC7,
  0xFF, 0xFF, 0xFF, 0xFF, 0xF7, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF1, 0xF3,
  0xC3, 0xFF, 0xF1, 0xC0, 0x99, 0x8F, 0xE7, 0xFF, 0xC0, 0x99, 0x8F, 0x99, 0xC1,
  0xF1, 0xFF, 0xE0, 0xC3, 0xFF, 0xFF, 0xC3, 0xFF, 0xFF, 0x9E, 0x99, 0xF3, 0xE3,
  0xF1, 0xE3, 0xFF, 0xFF, 0xFF, 0x81, 0xE1, 0xE3, 0xE7, 0xFF, 0xFF, 0x9E, 0x9E,
  0xEE, 0xD5, 0x92, 0xF3, 0xF3, 0xF3, 0xE4, 0xFF, 0xFF, 0xE4, 0xE4, 0xFF, 0xE4,
  0xF3, 0xF3, 0xFF, 0xF3, 0xFF, 0xF3, 0xF3, 0xE4, 0xE4, 0xFF, 0xE4, 0xFF, 0xE4,
  0xE4, 0xFF, 0xFF, 0xE4, 0xF3, 0xFF, 0xFF, 0xE4, 0xF3, 0xF3, 0xFF, 0x80, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0x81, 0xFF, 0xFF, 0xFF, 0x81, 0xE3, 0xE3, 0xF1, 0xFF,
  0xFF, 0xE7, 0xCF, 0xF3, 0xF8, 0xF3, 0xE7, 0xFF, 0xE3, 0xFF, 0xFF, 0xF8, 0xC3,
  },
// Row 2
 {
  0xFF, 0xBF, 0x80, 0x80, 0xE3, 0xC1, 0xF7, 0xFF, 0x80, 0xE1, 0x9E, 0xFC, 0xCC,
  0x8F, 0xF8, 0xE1, 0xCC, 0x92, 0xCE, 0xFF, 0xE1, 0xE1, 0xF3, 0xF3, 0xE7, 0xFF,
  0xFF, 0xC3, 0xC9, 0xC9, 0xC1, 0x9C, 0xC9, 0xCF, 0xE7, 0xE7, 0xCC, 0xE7, 0xFF,
  0x9C, 0xC7, 0x99, 0x99, 0xE1, 0x9F, 0xCF, 0x99, 0x99, 0x99, 0xE7, 0xE7, 0xE7,
  0x9C, 0xC3, 0xCC, 0xCC, 0xC9, 0xCE, 0xCE, 0xCC, 0x99, 0xE7, 0xF9, 0xCC, 0xCF,
  0xCC, 0x99, 0xCC, 0x99, 0xA5, 0x99, 0x99, 0x9C, 0x9C, 0x99, 0x9C, 0xCF, 0xCF,
  0xE7, 0xFF, 0xCF, 0xFF, 0xF9, 0xFF, 0xC9, 0xFF, 0xCF, 0xFF, 0xFF, 0xCF, 0xE7,
  0xFF, 0xFF, 0xFF, 0xFF, 0xE7, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xE7, 0xF3,
  0x99, 0x99, 0xFF, 0x9E, 0xFF, 0xFF, 0xE7, 0xFF, 0x9E, 0xFF, 0xFF, 0xFF, 0x9C,
  0xFF, 0xFF, 0xC9, 0x99, 0x99, 0x8F, 0x99, 0x8F, 0x99, 0xF3, 0xFF, 0xF3, 0xC9,
  0xFF, 0xFF, 0xF1, 0xF1, 0x83, 0xFF, 0xC9, 0xC9, 0xFF, 0xFF, 0xFF, 0x9C, 0x9C,
  0xBB, 0xAA, 0xC4, 0xF3, 0xF3, 0xF3, 0xE4, 0xFF, 0xFF, 0xE4, 0xE4, 0xFF, 0xE4,
  0xF3, 0xF3, 0xFF, 0xF3, 0xFF, 0xF3, 0xF3, 0xE4, 0xE4, 0xFF, 0xE4, 0xFF, 0xE4,
```

```
  0xE4, 0xFF, 0xFF, 0xE4, 0xF3, 0xFF, 0xFF, 0xE4, 0xF3, 0xF3, 0xFF, 0x80, 0xFF,
  0xFF, 0xC3, 0x81, 0x80, 0x99, 0xFF, 0xCC, 0xC4, 0xE7, 0xC9, 0xC9, 0xE7, 0xFF,
  0x81, 0xE7, 0xE7, 0xE7, 0xF2, 0xF3, 0xE7, 0xC4, 0xC9, 0xFF, 0xFF, 0xF9, 0xC9,
  },
// Row 3
 {
  0xFF, 0xAD, 0x92, 0x80, 0xC1, 0xE3, 0xE3, 0xF3, 0x8C, 0xCC, 0xB3, 0xF8, 0xCC,
  0x83, 0xE0, 0xC0, 0xCC, 0x92, 0xE3, 0xFF, 0xC0, 0xC0, 0xF3, 0xF9, 0xCF, 0x9F,
  0xFF, 0xC3, 0xC9, 0x80, 0x9F, 0x99, 0xE3, 0x9F, 0xCF, 0xF3, 0xE1, 0xE7, 0xFF,
  0x98, 0xE7, 0xF9, 0xF9, 0xC9, 0x83, 0x9F, 0xF9, 0x99, 0x99, 0xE7, 0xE7, 0xCF,
  0x90, 0x99, 0xCC, 0x9F, 0xCC, 0xCB, 0xCB, 0x9F, 0x99, 0xE7, 0xF9, 0xC9, 0xCF,
  0xCC, 0x99, 0xCC, 0x8F, 0xE7, 0x99, 0x99, 0x9C, 0xC9, 0x99, 0xB9, 0xCF, 0xE7,
  0xF3, 0xC3, 0xCF, 0xC3, 0xF9, 0xC3, 0xCF, 0xC4, 0xC9, 0xC7, 0xF9, 0xCC, 0xE7,
  0x91, 0xC4, 0x91, 0xC1, 0xC1, 0x99, 0x99, 0x9C, 0x9C, 0x99, 0x81, 0xE7, 0xF3,
  0x9F, 0xFF, 0xC3, 0xE1, 0xC3, 0xC3, 0xC3, 0xC3, 0xE1, 0xC3, 0xC3, 0xC7, 0xE3,
  0x81, 0xC0, 0x99, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xE1, 0x99, 0xC0, 0xCD,
  0xC3, 0xC7, 0xFF, 0xFF, 0xFF, 0x99, 0xC9, 0xC9, 0xE7, 0xFF, 0xFF, 0x99, 0x99,
  0xEE, 0xD5, 0x92, 0xF3, 0xF3, 0x83, 0xE4, 0xFF, 0x83, 0x84, 0xE4, 0x80, 0x84,
  0xF3, 0xF3, 0xFF, 0xF3, 0xFF, 0xF3, 0xF0, 0xE4, 0xE4, 0xE0, 0x84, 0x80, 0xE4,
  0xE4, 0x80, 0xFF, 0xE4, 0xF0, 0xF0, 0xFF, 0xE4, 0x80, 0xF3, 0xFF, 0x80, 0xFF,
```

To compile you also need the timer helpers library:

The TimerHelpers.h file can be downloaded from:

http://gammon.com.au/Arduino/TimerHelpers.zip

Communicating with the VGA sketch

The problem with this sketch is, it's very busy. Almost all its time is spent refreshing the screen. There is a horizontal sync pulse every 32 µs and then it spends 25 µs drawing the pixels. So, not much time over to do other stuff. (There's a bit, because during the vertical retrace we aren't drawing pixels).

So really, we need to have a second processor that actually does the useful work (eg. monitor the aquarium, do the burglar alarm, measure the temperature) and send the text to the VGA output sketch for displaying.

To achieve this, the only real way is to use I2C. A pin needed for SPI (pin 10) is in use by Timer 1 for Vsync. The serial hardware is in use by the output routines. So that leaves I2C.

In the "communicating" version I have set up a I2C request event, like this:

```
  // for incoming data to display from I2C
  Wire.begin (i2cAddress);
  Wire.onReceive (receiveEvent);
```

The slave address is currently 42, but you can change that to any free address (just make sure the sending end uses the same address).

When incoming I2C data arrives it is directed to a "state machine" that decodes the text and puts it into the "message" array (30 lines x 20 characters). To make it more useful you can send a few "control" characters, namely:

- carriage-return (0x0D): returns cursor to start of current line
- newline (0x0A): drops down a line and also goes to the start of the line
- clear screen (0x0C): clear screen, return cursor to 1,1

- ESC (0x1B) followed by:
    - 1 : clear screen, return cursor to 1,1
    - 2 : clear to end of current line
    - 3 : go to x,y ... next two bytes are X and then Y: one-relative

The newline character also scrolls the display, by moving everything up one line, if it occurs with the cursor on the 30th line.

Unfortunately incoming data can tend to make the display "glitch" as the interrupt stops the display drawing in the middle of a line. Attempts to correct that have not been a big success. If you turn off interrupts long enough to stop the glitching you then have the problem of incoming data being dropped. So I think it's better to have the occasional glitch, and see everything, than have important messages disappear.

To try to minimize this I increased the I2C clock rate in the sending "test" sketch, so that we transfer the message to the VGA output sketch as fast as possible.

Example sending sketch:

```
// Video text sender
// Author: Nick Gammon
// Date:    20th April 2012
// License: Released to public domain.

#include <Wire.h>

const byte i2cAddress = 42;

enum SEND_COMMANDS { CLRSCR = 1, CLREOL, GOTOXY, ESC = 27 };
int count = 0;

char clearScreen [] = { ESC, CLRSCR };
char clearToEndOfLine [] = { ESC, CLREOL };
char gotoXY [] = { ESC, GOTOXY };  // followed by x and y

void setup ()
{
  Wire.begin ();
  TWBR = 4;   // fast .. fast .. I2C : 16000000 / (16 + 2 * 4) = 666666 Hz
}  // end of setup

void sendString (const char * s)
  {
  Wire.beginTransmission (i2cAddress);
  Wire.write (s);
  Wire.endTransmission ();
  }  // end of sendString

char buf [20];

void loop ()
{
  sendString (clearScreen);
  sendString ("Hi there\n");
  sendString ("Everyone\n");

  delay (2000);

  // clear screen test
  sendString (clearScreen);
  sendString ("After CLRSCR\n");

  // display a counter
  sprintf (buf, "%i\n", count++);

  sendString (buf);

  delay (2000);
}  // end of loop
```

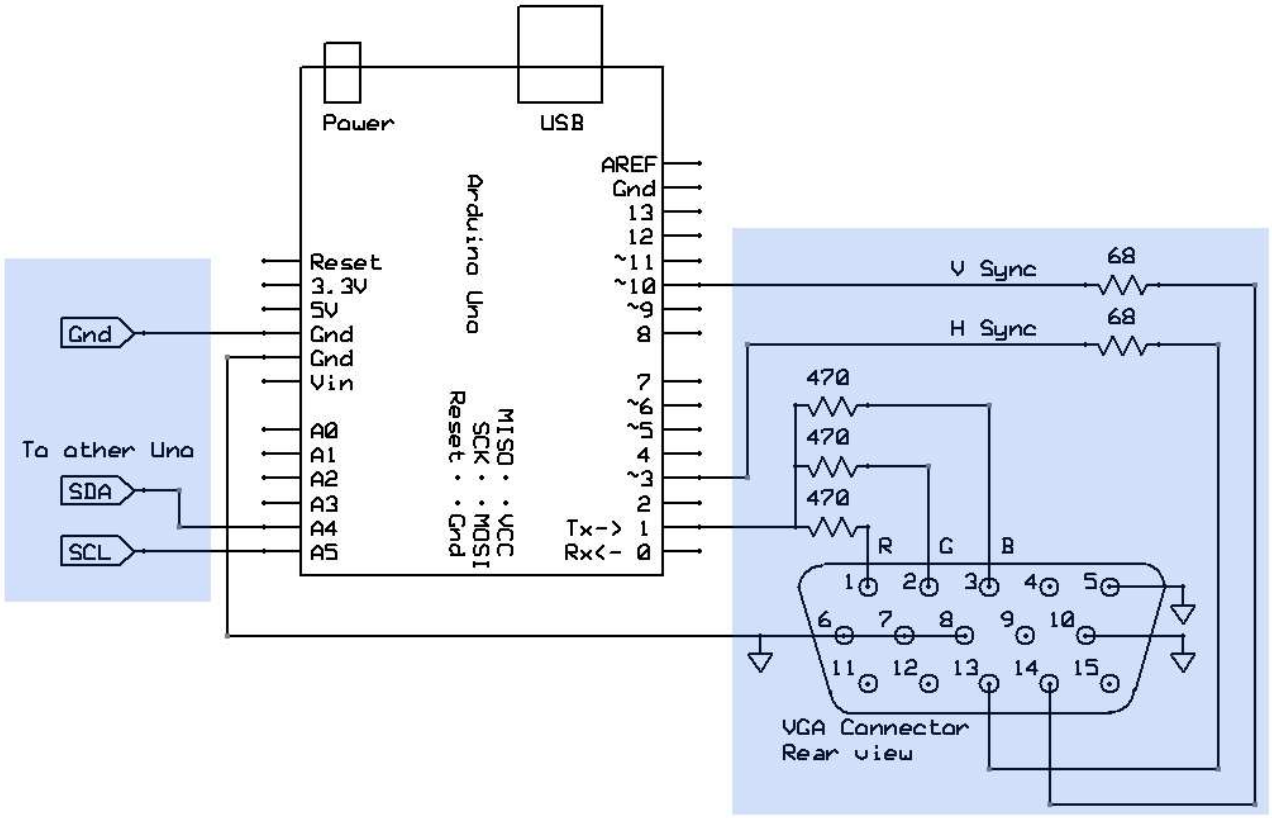
To save a lot of repetition the sending is done inside the sendString function, which does the begin and end transmission as well. Note that I2C has a buffer size of 32, so don't try to send more than 32 bytes in one send. (So, a line at a time would be a good compromise).

Code with communications stuff added

```
/*
 VGA video generation

 Author:   Nick Gammon
 Date:     20th April 2012
 Version:  1.2

 Version 1.0: initial release
 Version 1.1: code cleanups
 Version 1.2: more cleanups, added clear screen (0x0C), added scrolling


 Connections:

 D1 : Pixel output (470 ohms in series to each one of R, G, B)   --> Pins 1, 2,
 D3 : Horizontal Sync (68 ohms in series) --> Pin 13 on DB15 socket
 D10 : Vertical Sync (68 ohms in series) --> Pin 14 on DB15 socket

 Gnd : --> Pins 5, 6, 7, 8, 10 on DB15 socket

 PERMISSION TO DISTRIBUTE

 Permission is hereby granted, free of charge, to any person obtaining a copy of
 and associated documentation files (the "Software"), to deal in the Software wi
 including without limitation the rights to use, copy, modify, merge, publish, d
 and/or sell copies of the Software, and to permit persons to whom the Software
 subject to the following conditions:

 The above copyright notice and this permission notice shall be included in
 all copies or substantial portions of the Software.


 LIMITATION OF LIABILITY

 The software is provided "as is", without warranty of any kind, express or impl
 including but not limited to the warranties of merchantability, fitness for a p
 purpose and noninfringement. In no event shall the authors or copyright holders
 for any claim, damages or other liability, whether in an action of contract,
 tort or otherwise, arising from, out of or in connection with the software
 or the use or other dealings in the software.

*/

#include <TimerHelpers.h>
#include <avr/pgmspace.h>
#include "screenFont.h"
#include <avr/sleep.h>
#include <Wire.h>

#define BETA_ARDUINO ARDUINO < 100

const byte pixelPin = 1;     // <------- Pixel data
const byte hSyncPin = 3;     // <------- HSYNC
const byte MSPIM_SCK = 4;    // <-- we aren't using it directly
const byte vSyncPin = 10;    // <------- VSYNC

const int horizontalBytes = 20;  // 160 pixels wide
const int verticalPixels = 480;  // 480 pixels high

const byte i2cAddress = 42;

// Timer 1 - Vertical sync
```

Wiring

This shows how to connect up the DB15 connector (socket) that you plug your VGA cable into:



The 470 ohm resistors are there to convert the 5V output from the Uno into 0.7V for the analog pixel data. This is because there is a 75 ohm resistor inside the monitor, so the 470 ohm resistors form voltage dividers:

```
75 / (75 + 470) * 5 = 0.688 V
```

688 mV is close enough to the required 700 mV.

I'm not sure about the 68 ohms resistors for the two TTL lines (sync lines) but I have seen them suggested on other web sites.

Of course, if you prefer green or blue output, just omit the wire to the appropriate pins. You can have red/green/blue or some combination (cyan/magenta/yellow/white).

The three wires on the left are for connecting up to another processor to feed this one with the data to be displayed.

Download

The VGA Output sketch above, plus the font file, and the timers helper file, are all in the .zip file here:

http://gammon.com.au/Arduino/VGA_output.zip

Where to go from here? Colour?

I've had suggestions to "improve" the sketch by adding colour. Hopefully I've explained why you can't simply do that, without sacrificing something. The output loop is already very tight, with only one clock cycle to spare.

Let's say, for example, that you wanted to send colour. First you would need to have 3 pixel pins (reg/green/blue), so that rules out using SPI. You would have to manually load up (bit bang) a register with all 3 colours. This would be a lot slower. Also you would need to keep the colour information somewhere.

And to do graphics you would need to keep the "bitmap" somewhere. Even to store the bits for 160 x 480 x 4 bit colour would require 160 * 480 * 4 bits = 307200 bits which is 38400 bytes. Well there simply isn't that much memory in the processor. You have 2048 bytes.

Summary of timings

```
Vertical

60 Hz frame rate
1/60 = 0.016667 Seconds period (16667 µs)

525 vertical scan lines (480 visible)

Therefore:  1 / 60 / 525 = 31.746 µs per line (13.5 KHz)

Vertical sync pulse: 2 lines (   64 µs)
Back porch:         33 lines ( 1047 µs)
Visible area:      480 lines (15238 µs)
Front porch:        10 lines (  317 µs)
---------------------------------------
TOTAL                          16666 µs per frame


Horizontal

31.746 µs per line (13.5 KHz)

800 pixels per line (640 visible)

Therefore: 1 / 60 / 525 / 800 = 39.68 ns per pixel (25.2 MHz)

Horizontal sync pulse: 96 pixels ( 3.81 µs)
Back porch:            48 pixels ( 1.90 µs)
Visible area:         640 pixels (25.40 µs)
Front porch:           16 pixels ( 0.63 µs)
-----------------------------------------
TOTAL                               31.74 µs per line
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,775 posts) bio *Forum Administrator*

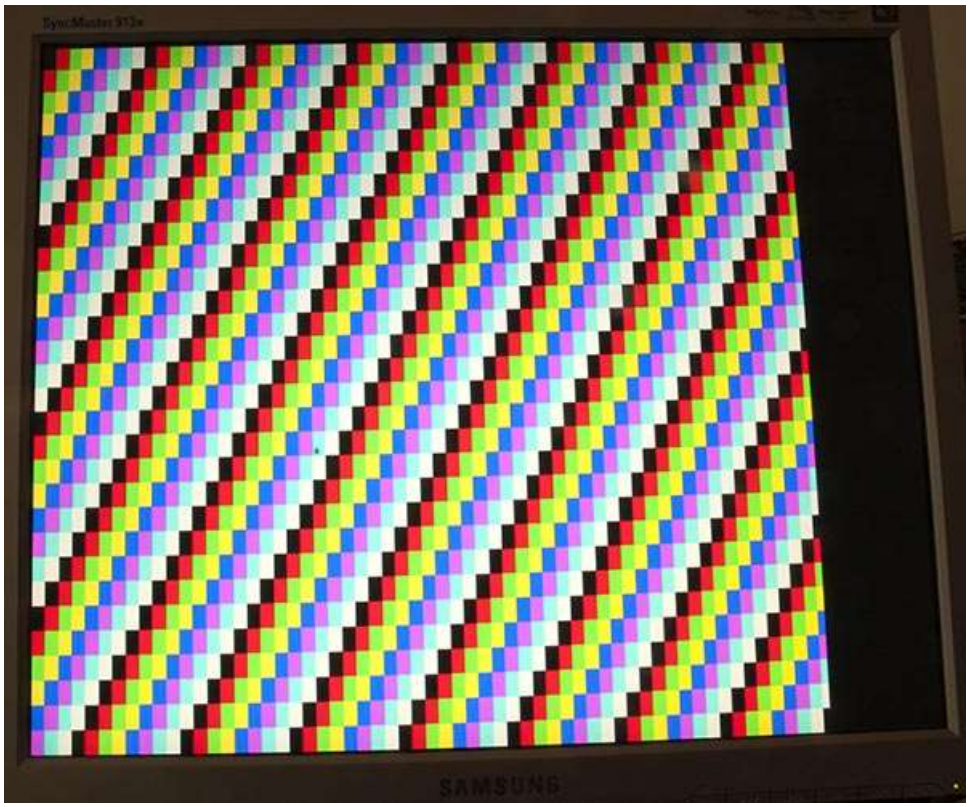
Date Reply #1 on Sun 22 Apr 2012 01:09 AM (UTC)

Amended on Tue 26 Nov 2013 01:38 AM (UTC) by Nick Gammon

Message

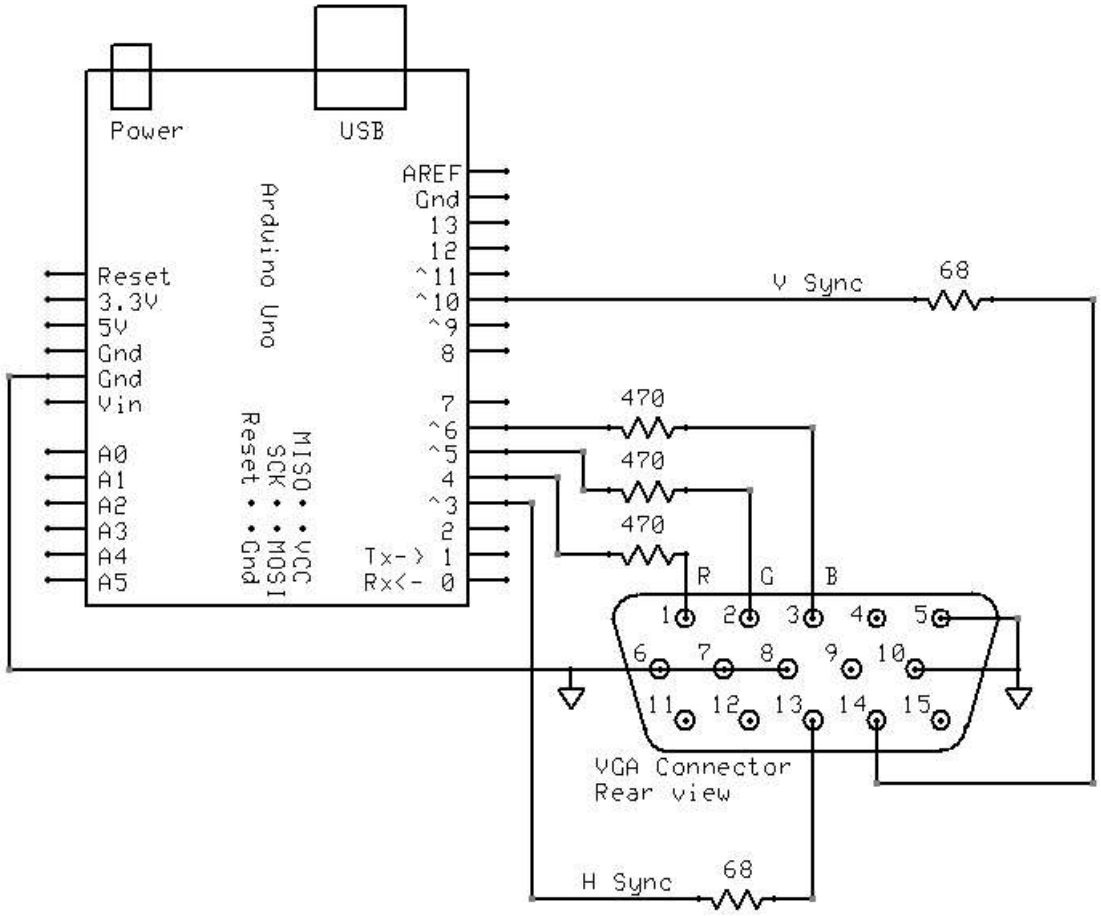
Colour output

Just as an experiment, I modified the above sketch to try to output colour. Here is the result:



Wiring for colour

For colour output we need to send a different signal to the R, G and B pins, like this:



Colour output sketch

```
  // Timer 2 - horizontal sync pulses
  pinMode (hSyncPin, OUTPUT);
  Timer2::setMode (7, Timer2::PRESCALE_8, Timer2::CLEAR_B_ON_COMPARE);
  OCR2A = 63;   // 32 / 0.5 uS = 64 (less one)
  OCR2B = 7;    // 4 / 0.5 uS = 8 (less one)
  TIFR2 = bit (TOV2);   // clear overflow flag
  TIMSK2 = bit (TOIE2);  // interrupt on overflow on timer 2

  // prepare to sleep between horizontal sync pulses
  set_sleep_mode (SLEEP_MODE_IDLE);

  // pins for outputting the colour information
  pinMode (redPin, OUTPUT);
  pinMode (greenPin, OUTPUT);
  pinMode (bluePin, OUTPUT);

}  // end of setup

// draw a single scan line
void doOneScanLine ()
  {

  // after vsync we do the back porch
  if (backPorchLinesToGo)
    {
    backPorchLinesToGo--;
    return;
    }  // end still doing back porch

  // if all lines done, do the front porch
  if (vLine >= verticalPixels)
    return;

  // pre-load pointer for speed
  register char * messagePtr =  & (message [messageLine] [0] );
```

```
  delayMicroseconds (1);

  // how many pixels to send
  register byte i = horizontalBytes;

  // blit pixel data to screen
  while (i--)
    PORTD = * messagePtr++;

  // stretch final pixel
  nop; nop; nop;

  PORTD = 0;  // back to black

  // finished this line
  vLine++;

  // every 16 pixels it is time to move to a new line in our text
  if ((vLine & 0xF) == 0)
    messageLine++;

  }  // end of doOneScanLine

void loop()
  {
  // sleep to ensure we start up in a predictable way
  sleep_mode ();
  doOneScanLine ();
}  // end of loop
```

Main output loop

The code to output the pixels is a simple loop:

```
while (i--)
    PORTD = * messagePtr++;
```

It relies on the pixel data pins all being on the same hardware port (PORTD in this case).

The generated assembler code is:

```
  while (i--)
    PORTD = * messagePtr++;
(2) 194:        89 91           ld      r24, Y+
(1) 196:        8b b9           out     0x0b, r24       ; 11
(1) 198:        91 50           subi    r25, 0x01       ; 1
(2) 19a:        e0 f7           brcc    .-8             ; 0x194

-------
6 cycles in loop = 375 nS
```

Thus it takes 6 cycles (375 nS) per pixel, and the 60 pixels will take 22.5 uS which is inside the 25.4 uS allowed time for the visible frame.

Notes

It turned out we had time to send 60 pixels horizontally. The width of each pixel is entirely due to the time it takes to read the next one from memory and clock it out (6 clock cycles).

Vertically we draw 30 pixels (each one is drawn for 16 scan lines, taking a total of 480 pixels from the monitor's point of view). The bitmap alone takes 30 * 60 = 1800 bytes. That only leaves 248 bytes for the rest of the program, which (once you allow for the fact that around the first 100 are used by the processor for control registers) doesn't give much over.

The pixels are copied from the "message" array which is initialized into a nice checkerboard pattern in setup. You could change that (if you can find some free cycles) to animate graphics.

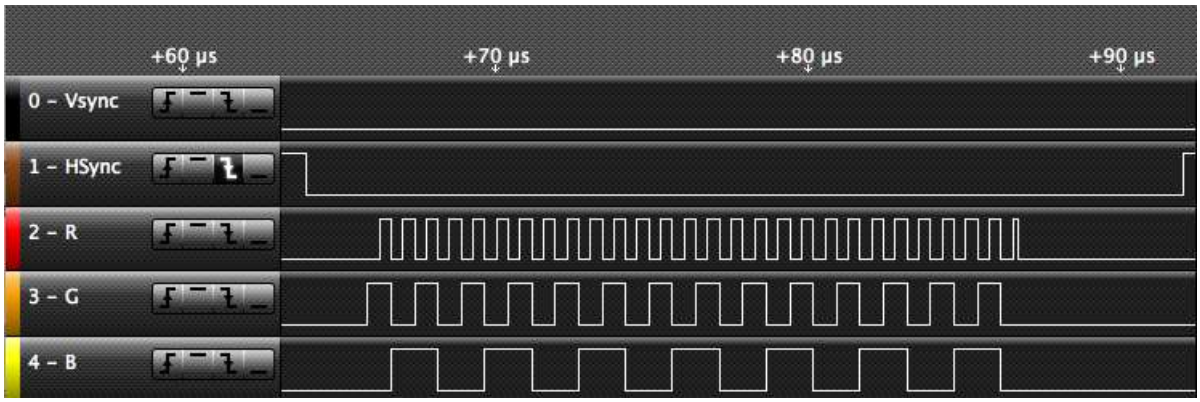
Be warned that there are about 34 bytes of SRAM free ... not a lot. If you need more memory you could compromise by making the pixels bigger, so not as much memory is needed to hold them.

Final pixel

After taking the above photo, I realized the final pixel was a bit narrow. This is because we change to black a bit too fast. The amended code above now has three "nop" (no operation) instructions added to stretch that final pixel a bit.

Logic analyzer output

This shows the three different colours changing in one scan line:



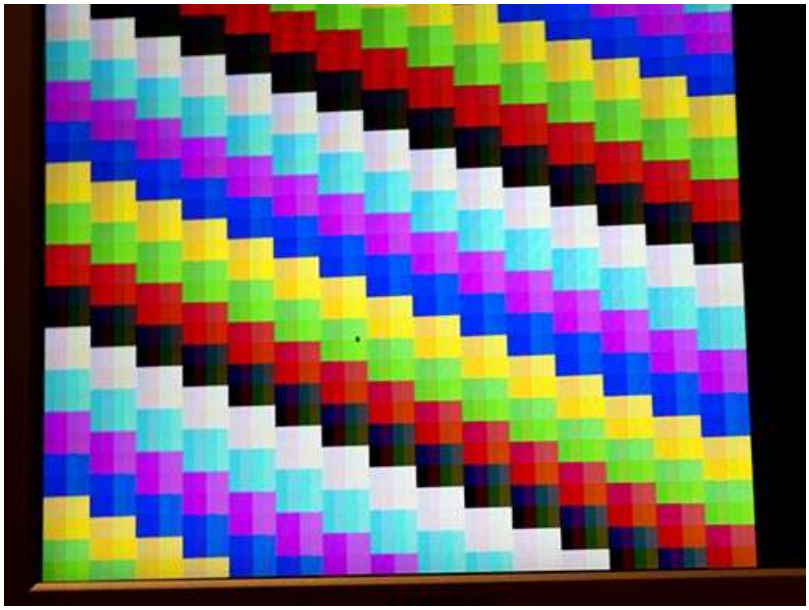
More colours

You could have more colours for the same amount of memory, and the same speed, by using 6 pins (2 for each colour) with a different value resistor for the second 3 pins. I haven't done the maths, but something like 1K resistors in series for the second set of pins would mean if they were active you would have half the voltage (eg. 0.35V) of video rather than 0.7V). Thus you could have "dull" and "bright" for each of R, G, B and of course all sorts of combinations inbetween.

The pins would need to be chosen from PORTD still, for example D0, D1, D2. Since we are already copying a single byte to PORTD we could simply have more "active" bits in that byte. That would give 64 colours (2^6) rather than 8 colours (2^3).

Example of 64 colours

This is an example of using more colours:



That was produced by this sketch:

```
/*
 VGA colour video generation

 Author:   Nick Gammon
 Date:     22nd April 2012
 Version:  1.0

 Version 1.0: initial release
 Version 1.1: Amended to output 64 colours

 Connections:

 D0 : Dull Red pixel output (1K resistor in series) --> Pin 1 on DB15 socket
 D1 : Dull Green pixel output (1K resistor in series) --> Pin 2 on DB15 socket
 D2 : Dull Blue pixel output (1K resistor in series) --> Pin 3 on DB15 socket
 D3 : Horizontal Sync (68 ohms in series) --> Pin 13 on DB15 socket
 D4 : Red pixel output (470 ohms in series) --> Pin 1 on DB15 socket (also)
 D5 : Green pixel output (470 ohms in series) --> Pin 2 on DB15 socket (also)
 D6 : Blue pixel output (470 ohms in series) --> Pin 3 on DB15 socket (also)
 D10 : Vertical Sync (68 ohms in series) --> Pin 14 on DB15 socket

 Gnd : --> Pins 5, 6, 7, 8, 10 on DB15 socket


 Note: As written, this sketch has 34 bytes of free SRAM memory.

 PERMISSION TO DISTRIBUTE

 Permission is hereby granted, free of charge, to any person obtaining a copy of
 and associated documentation files (the "Software"), to deal in the Software wi
 including without limitation the rights to use, copy, modify, merge, publish, d
 and/or sell copies of the Software, and to permit persons to whom the Software
 subject to the following conditions:

 The above copyright notice and this permission notice shall be included in
 all copies or substantial portions of the Software.


 LIMITATION OF LIABILITY

 The software is provided "as is", without warranty of any kind, express or impl
 including but not limited to the warranties of merchantability, fitness for a p
 purpose and noninfringement. In no event shall the authors or copyright holders
 for any claim, damages or other liability, whether in an action of contract,
 tort or otherwise, arising from, out of or in connection with the software
 or the use or other dealings in the software.

*/
```

```
#include <TimerHelpers.h>
#include <avr/pgmspace.h>
#include <avr/sleep.h>

const byte hSyncPin = 3;     // <------- HSYNC

const byte redPin = 4;       // <------- Red pixel data
const byte greenPin = 5;     // <------- Green pixel data
const byte bluePin = 6;      // <------- Blue pixel data

const byte dullRedPin = 0;     // <------- Red pixel data 50% brightness
const byte dullGreenPin = 1;   // <------- Green pixel data 50% brightness
const byte dullBluePin = 2;    // <------- Blue pixel data 50% brightness
```

I connected the red/green/blue pins on the DB15 to Arduino pins D0/D1/D2 via 1K resistors.

Example of moving graphics

VGA Output on Arduino

Nick Gammon

00:28

Code to produce:

```
/*
 VGA colour video generation - Sine wave generation

 Author:   Nick Gammon
 Date:     22nd April 2012
 Version:  1.0

 Version 1.0: initial release

 Connections:

 D3 : Horizontal Sync (68 ohms in series) --> Pin 13 on DB15 socket
 D4 : Red pixel output (470 ohms in series) --> Pin 1 on DB15 socket
 D5 : Green pixel output (470 ohms in series) --> Pin 2 on DB15 socket
 D6 : Blue pixel output (470 ohms in series) --> Pin 3 on DB15 socket
 D10 : Vertical Sync (68 ohms in series) --> Pin 14 on DB15 socket
```

```
  Gnd : --> Pins 5, 6, 7, 8, 10 on DB15 socket


  Note: As written, this sketch has 34 bytes of free SRAM memory.

  PERMISSION TO DISTRIBUTE

  Permission is hereby granted, free of charge, to any person obtaining a copy of
  and associated documentation files (the "Software"), to deal in the Software wi
  including without limitation the rights to use, copy, modify, merge, publish, d
  and/or sell copies of the Software, and to permit persons to whom the Software
  subject to the following conditions:

  The above copyright notice and this permission notice shall be included in
  all copies or substantial portions of the Software.


  LIMITATION OF LIABILITY

  The software is provided "as is", without warranty of any kind, express or impl
  including but not limited to the warranties of merchantability, fitness for a p
  purpose and noninfringement. In no event shall the authors or copyright holders
  for any claim, damages or other liability, whether in an action of contract,
  tort or otherwise, arising from, out of or in connection with the software
  or the use or other dealings in the software.

*/

#include <TimerHelpers.h>
#include <avr/pgmspace.h>
#include <avr/sleep.h>

const byte hSyncPin = 3;     // <------- HSYNC

const byte redPin = 4;       // <------- Red pixel data
const byte greenPin = 5;     // <------- Green pixel data
const byte bluePin = 6;      // <------- Blue pixel data

const byte vSyncPin = 10;    // <------- VSYNC

const int horizontalBytes = 50;  // 480 pixels wide
const int verticalPixels = 480;  // 480 pixels high

// Timer 1 - Vertical sync
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,775 posts) bio *Forum Administrator*

Date Reply #2 on Sat 07 Feb 2015 11:47 PM (UTC)

Amended on Sun 08 Feb 2015 10:28 AM (UTC) by Nick Gammon

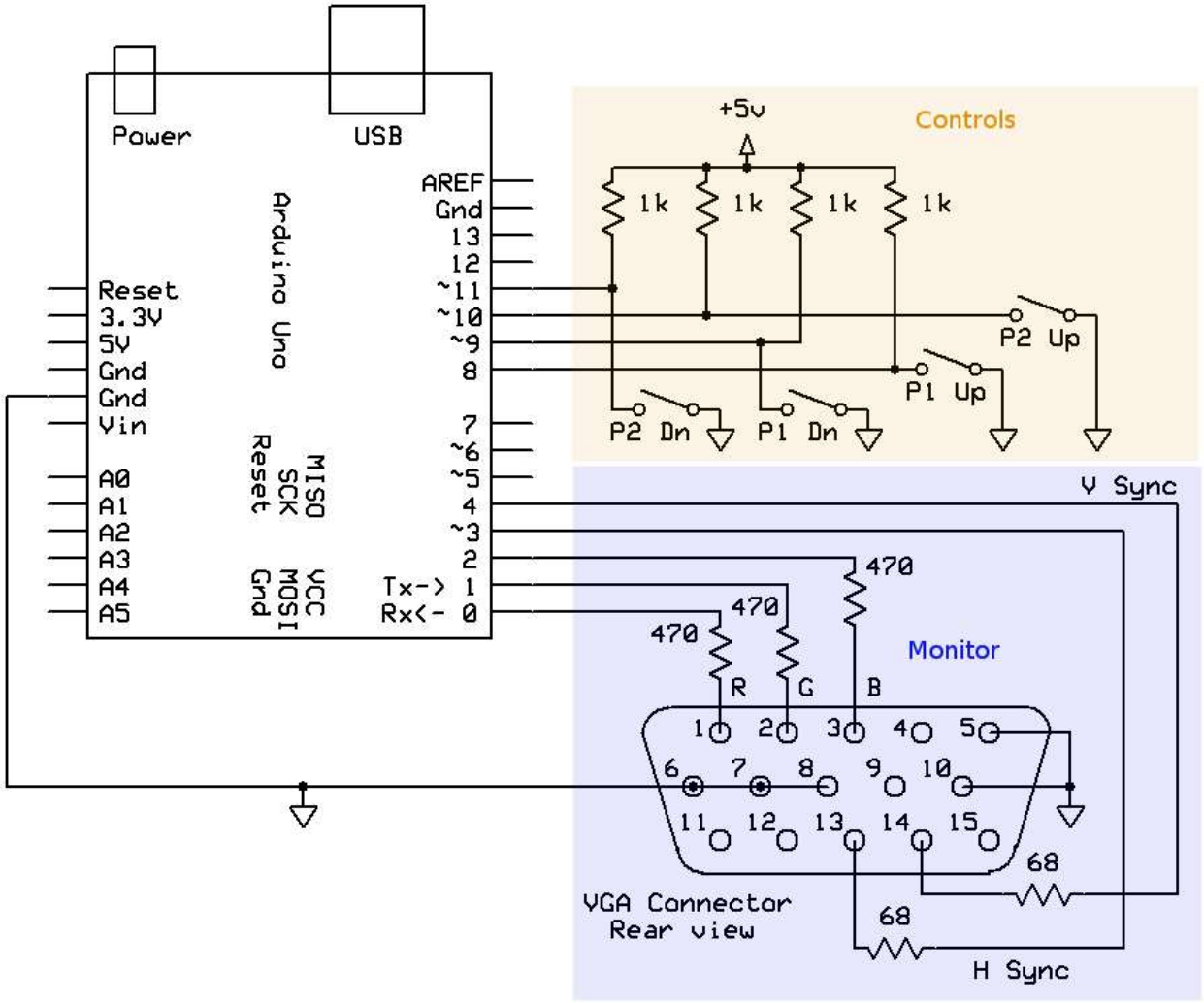
Message

Pong game using VGA output

Following up from the above posts, an Arduino forum user "LosTZealoT" (Michael Rosen) posted code (here) to make a "Pong" game using the same basic idea, but only using one timer.

Schematic

The wiring for his version is:



Description

From Michael's post:

> I liked your approach but thought it could be interesting to get a bitmapped version that used less resources (you use two timers and the main loop D:).
>
> So I thought I'd try my hand at it and came up with this. It only uses Timer2 and by doing everything in the Timer2 ISR, it saves the loop function for things like games or what not (mine is bitmapped). My resolution is terrible, 40x30, but this is primarily due to memory constraints as I wanted a multiple of 640x480 that would fit in the Arduino UNO's RAM. I havent looked into making the "screen" bigger but it is likely possible. I did expand it to your 2-bit per channel color as well, here it is ...

Code

```
break;
```

```
  case P2_SCORE:
    /* Give P2 a point, move ball back to start and its P1's serve */
    p2Score++;
    gameState = P1_SERVE;
    ballY = (BOTTOM - TOP - BALL_HEIGHT) / 2;
    ballX = (P2_SIDE - P1_SIDE - BALL_WIDTH) / 2;
    ballDx = 0;

    ballDy = 0;
    break;
  case PLAYING:
    /* Check to see if a player scored, if so, moved to that state */
    if (ballX <= P1_SIDE) {
      gameState = P2_SCORE;
    }
    else if (ballX >= P2_SIDE) {
      gameState = P1_SCORE;
    }
    break;
  }

  /* Paddle movement */
  if (p1Up && (p1Y - PADDLE_SPEED) >= TOP) {
    p1Y -= PADDLE_SPEED;
  }
  else if (p1Down && (p1Y + PADDLE_SPEED) <= (BOTTOM - PADDLE_HEIGHT + 1)) {
    p1Y += PADDLE_SPEED;
  }

  if (p2Up && (p2Y - PADDLE_SPEED) >= TOP) {
    p2Y -= PADDLE_SPEED;
  }
  else if (p2Down && (p2Y + PADDLE_SPEED) <= (BOTTOM - PADDLE_HEIGHT + 1)) {
    p2Y += PADDLE_SPEED;
  }

  /* Ball Movement */
  if (((ballY + ballDy) < TOP) ||
    ((ballY + ballDy) > BOTTOM)) {
    ballDy = -ballDy;
  }
  if (((ballY + ballDy) >= p1Y) &&
    ((ballY + ballDy) < p1Y + PADDLE_HEIGHT) &&
    ((ballX + ballDx) >= P1_COL) &&
    ((ballX + ballDx) < P1_COL + PADDLE_WIDTH)) {
    ballDx = -ballDx;
  }
  else if (((ballY + ballDy) >= p2Y) &&
    ((ballY + ballDy) < p2Y + PADDLE_HEIGHT) &&
    ((ballX + ballDx) >= P2_COL) &&
    ((ballX + ballDx) < P2_COL + PADDLE_WIDTH)) {
    ballDx = -ballDx;
  }

  ballX += ballDx;
  ballY += ballDy;

  flipDir = -flipDir;

  /* Actual render the game */
  drawGame();

  delay(100);
}
```

Code amended slightly by me (Nick Gammon) to allow for pull-up resistors for the controls, ran it through the IDE auto-format, and changed _BV() to bit().

It looks like Michael was using switching 5V to the controls. I tried with internal pull-ups but they didn't work, quite possibly from all the "noise" of the VGA output.

Michael explains:

> I'll admit the Timer2 COMPB ISR is weird, but thats because I thought I had a timing issue in it for the longest time. At least in its current form, it isnt terribly slow to start drawing. Also, it is interleaved to meet certain timing.
>
> All you need to due is set up screen to have the correct value, where bit 4 is 0. The way I had it hooked up, D0-7 where BrightRed (D0), BrightGreen (D1), BrightBlue (D2), Hsync (D3), Vsync (D4), DarkRed (D5), DarkGreen (D6), DarkBlue (D7), but other than the rearrangement, the connections matched your 2-bit color circuit.
>
> Basically, with this code, you can fill in loop with whatever code you need and by writing to screen, you get color. Heres PONG with my 1-bit color code (no score display though :( ):

Comments

Michael has cunningly managed to get VGA output with only one timer. The display is steady, without jitter. The game plays smoothly (as far as I can tell with my limited controls). The paddles are white and the ball is magenta.

Michael has achieved what I did not manage in my earlier example, to have useful user input at the same time as displaying VGA output. He has used PORTD (Arduino pins D0 to D7) to send all signals (red, green, blue, hsync and vsync) so that a single output to that port (effectively, two clock cycles) outputs the three colours and sync information at the same time.

In addition to that he uses PORTB (Arduino pins 8 to 13) to get player input from the game controls.

This is another brilliant example of what can be achieved with an Atmega328P, only using 2 kB of RAM and 32 kB of program memory. Another example (which uses more external hardware) is the Toorum's Quest game here: http://www.gammon.com.au/forum/?id=12623

State machines

If anyone wants an example of state machines, this code shows the technique. In the Timer 2 interrupt routine, the variable "displayState" keeps track of the current state.

The states are IN_DRAW, IN_VSYNC_PULSE and IN_VSYNC_FP. The current state controls what is done when the interrupt fires, and at the appropriate time the state is changed.

Also "gameState" keeps track of how the game is going.

Game states are P1_SERVE, P2_SERVE, P1_SCORE, P2_SCORE, PLAYING.

Screenshot

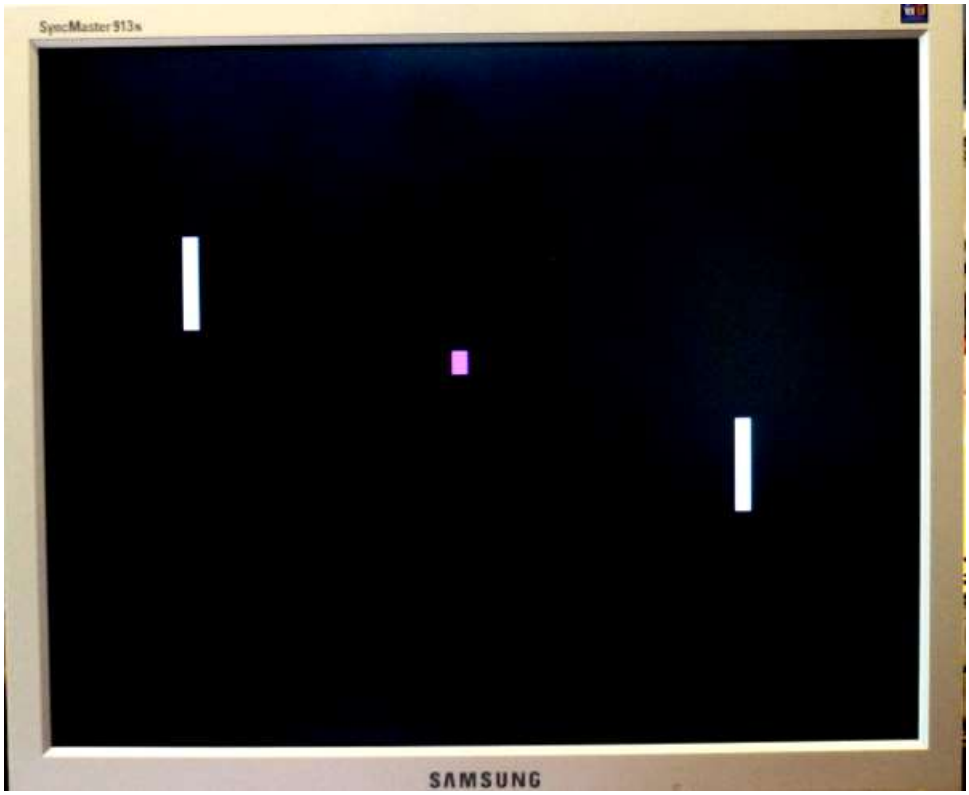


Photo courtesy of Isobel Gammon.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,775 posts) bio *Forum Administrator*

Date Reply #3 on Sat 02 May 2015 09:02 PM (UTC)

Amended on Sat 02 May 2015 09:05 PM (UTC) by Nick Gammon

Message

VGA library for Arduino UNO and ATMega328

Sandro Maffiodo has released a library based on the ideas in this thread to make a library that implements a 120 x 60 pixel frame-buffer where each pixel is stored as 2 bits (4 colours).

The frame-buffer is stored inside SRAM.

Post about it: VGA library for Arduino UNO and ATMega328

Example of it in operation:

Source code and description: VGAX Library for Arduino UNO

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,775 posts) bio *Forum Administrator*

Date Reply #4 on Sun 07 Jun 2015 09:15 PM (UTC)

Amended on Sun 07 Jun 2015 09:16 PM (UTC) by Nick Gammon

Message

Colour 'Space Invaders' style game for the Arduino

Rob Cai has made a colour game in the style of "Space Invaders" for the Arduino Uno or similar. See the video:

Arduino Bomber with Colors and Sound

It used the "VGAX Library for Arduino UNO" (see previous post), a few resistors, potentiometer, and DB15 connector. No special shields or support chips.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

The dates and times for posts above are shown in Universal Co-ordinated Time (UTC).

To show them in your local time you can join the forum, and then set the 'time correction' field in your profile to the number of hours difference between your location and UTC time.

153,839 views.

Postings by administrators only.

Refresh page

Go to topic: (Choose topic) Go Search the forum

top

Quick links: **MUSHclient**. MUSHclient **help**. Forum **shortcuts**. Posting **templates**. Lua **modules**. Lua **documentation**.

Home

Designed & written by Nick Gammon

Comments to: Gammon Software support

XML Forum RSS feed (https://gammon.com.au/rss/forum.xml)

Best Viewed With AnyBrowser

FutureQuest