AVR Video Generator with an AVR Mega163

POP QUIZ: WHAT DO YOU GET WHEN YOU MIX A SMALL BLACK AND WHITE TV WITH THE ATMEL AVR MEGA163? IF YOU'RE A LAB-ORIENTED PROFESSOR, YOU GET A STANDARDIZED, COST-EFFECTIVE VIDEO GENERATOR FOR YOUR CLASSROOM. FOLLOW ALONG AS BRUCE DESCRIBES HOW HE DID IT.

By: Bruce Land

Many interesting microcontroller applications have a graphics component. The students in my class (ECE 476 at Cornell University) often want to use graphics for their design projects. Over the past few years, the students have experimented with graphics devices including LCDs, arrays of LEDs, analog oscilloscopes used as x-y plotters, mechanical x-y plotters, and applications running under Windows. While many of the applications have been clever and worked well, none of the graphics devices were appropriate to deploy widely in a teaching setting.

Any device used in the teaching lab, whether it's a graphics device or microcontroller development board, must satisfy several criteria. First, the device must be relatively inexpensive (there are about 75 students in my course and we need 30 to 50 sets of hardware). Second, the device must be robust so that it has a chance of surviving. Third, the device must be easy to understand so that it can be taught quickly.

And fourth, the device must be standardized so that it can be easily replaced.

After looking at the options for graphics devices, it looked like a small, cheap black and white television would be perfect. Not only does it meet all of the above constraints, the TV is also a good example of a hard real-time system. If the sync pulses don't arrive on time, the image jitters or breaks up. Because the microcontroller is directly driving the TV, timing errors show up as a degraded image, giving the students rapid, understandable feedback.

The Atmel AVR Mega163 has just enough memory and speed to generate a black and white video signal. The implementation broke down into three parts: sync generation, image display, and image content generation. All three parts ran on a single Mega163. For ease of teaching, I wanted as much of the code as possible to be in C, with little or no assembler. With careful attention to the CodeVision compiler, I was successful and used just a few lines of assembler. The time-critical image display code is as good in C as I could write in assembler, requiring four machine cycles per pixel. The only external components were three resistors and two diodes that formed the video DAC.

Before going into the implementation details, let's briefly review how a TV is controlled. I learned much of the following from four web sites.

Photo 1: The "Cornell ECE476" message and time in the lower right corner are computed by the program. The "Circuit Cellar," triangle, and dot are the result of commands issued to the Mega163 from HyperTerminal. The commands are: t0530CIRCUIT (text starting at x = 05, y = 30); t1538CELLAR; L055058501 (line x1 = 05, y1 = 50, x2 = 58, y2 = 50, color = 1); L055032701; L585032701; and p32601 (point x = 32, y = 60, color = 1).

I obtained a lot of useful information from the Stanford University web site on a page for course EE 281. Pascal Stang's lab assignment, titled "TV Paint," provided insight. [1] Also, Alberto Ricci Bitti's excellent projects offered many useful hints, including putting the CPU to sleep just before an interrupt in order to make the interrupt timing more reliable. [2] Glen Williamson's site provides well-explained diagrams of video signals. [3] To read about a range of video projects and code, I also checked out Rickard Gunée's web site. [4]

Video Signal Generation

The video signal I decided to implement was a simplified version of RS170 (NTSC-rate black and white) video. RS170 uses a scheme in which sync pulses are 0V, black is about 0.3V, and white is about 1V. Each line starts with a 5-µs sync pulse. This pulse causes the TV to reset the electron beam to the left edge of the screen. After another 5 µs, you can start to write out the image content for that line (for a maximum of about 50 µs). At the end of each field (frame), the electron beam has to be moved back to the upper left corner of the screen to start a new field. The vertical sync pulse consists of three consecutive lines of sync-level voltage interrupted by inverted horizontal sync pulses.

Full RS170 uses interlaced fields, in which odd lines are drawn and then even lines are filled in-between them. Interlacing is used to reduce flicker.

Reprinted with permission of:
Circuit Cellar®
Issue 150
January 2003

My main simplification was not making an interlaced picture, but rather drawing every field with exactly 262 lines (instead of the RS170 standard of 262.5 lines per field). Figure 1 shows the timing relationships and line numbers. The duration of each line was increased slightly so that each field was exactly 1/60 s. NTSC specifies 63.55 µs per line. My code produces 63.625-µs lines. The resulting signal displays correctly on every TV I have tried; the signal does not flicker and it records perfectly on a VCR.

Each field consists of 242 lines on which image content may be displayed and 20 lines dedicated to generating the correct vertical sync pulses. The lines from 243 to 247 should be at black level with no image content. Lines 248 to 250 generate the actual vertical sync pulse. Lines 251 to 262 should remain at black level. Because no image content is displayed during lines 242 to 262, and because image display is the most CPU/memory-intensive task, new image content may be computed during these lines to be displayed when RAM is dumped to the screen again in the next field.

Sync generation occurs in an interrupt service routine (ISR) triggered from AVR Timer 1, compare-match channel A. The channel A compare-match function also (optionally) zeroes the timer in hardware to ensure an accurate time base. With an 8-MHz crystal, the timer interrupt is triggered every 509 cycles for a period equal to the horizontal sync rate of 63.625 µs. It is essential that the ISR always be entered from the CPU sleep state so that the interrupt latency remains the same number of cycles. Normally the AVR interrupt latency varies by one or two cycles because instructions cannot be interrupted and are one to three cycles long (most often one cycle).

In the ISR, the horizontal and vertical sync pulses are generated and the line counters are updated. All of the logic for counting lines, inverting the horizontal sync to make vertical sync, and changing the I/O port pin are contained within the 5-µs horizontal sync pulse time (see Listing 1).

The main program initializes ports, timers, and static image material, and then goes into a while loop. The loop is repeated once per line and includes an assembler sleep command to suspend execution until the next interrupt. You may download a summary of the program functions from the Circuit Cellar ftp site.

After the sync interrupt, the Mega163 draws one line of the image and then goes back to sleep until the next line. Image content resides in 800 bytes of RAM of the Mega163. The 6400 bits are arranged as 100 lines of 64 binary-level pixels per line. With 8 bytes per line, I could convert a pixel position to the address of the appropriate image byte using shifts and adds rather than multiplies.

At the beginning of each line, 8 bytes were pre-fetched from RAM and sent to the registers. The eight registers were then clocked out of a port pin, bit by bit. I unrolled all of the loops so that the pixel rate would be constant. Inspection of the assembler code generated by the compiler showed four cycles per pixel. Therefore, at 8 MHz, a pixel would be 0.5/63.625 of a scan line, or about 0.7%. The 64 pixels per line thus fill the middle half of the screen.

Photo 2: This DLA program example takes 5547 s to compute. You can see the current free particle as a blur in the lower right corner of the screen.

To make the pixels almost square, I duplicated them vertically onto two lines so that the 100 vertical pixels covered 200 video scan lines. This duplication also reduced flickering.

To display an image, you need to compute points, lines, and text to fill the image RAM. I wrote a point plotter routine that could set, clear, or invert a pixel given its x, y coordinates. On top of the point routine, I wrote a Breshenham line drawing routine that runs quickly on a small CPU because it requires no divisions, only shifts and adds. [5] I wrote two character generators, one for 5 x 7 characters and one for 3 x 5 characters. The 3 x 5 characters are adequate for numbers, but marginal for text. However, the 3 x 5 characters can be drawn quickly because they're placed into image memory via a bit copy operation and do not use the point drawing routine. The 5 x 7 characters are placed into image memory point by point. The precise placement creates better-looking and denser text. On top of the character generators are unctions that take string input.

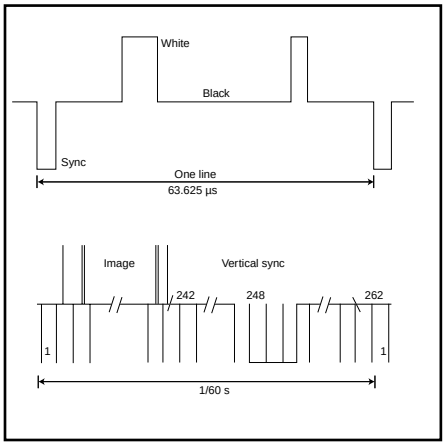

Figure 1: The top trace shows the waveform of one line of the video signal. Each horizontal sync pulse is 5-µs long. The sync level is 0 V, the black level is 0.3 V, and the white level is 1 V. The bottom trace shows the waveform of one field (frame) of the video. Each narrow downward pulse is a horizontal sync pulse. The line numbers within the field are numbered. Image data appears on lines one to 242. Vertical sync starts at line 243 and ends at line 262.

Two port pins are used: one for the video level and one for the sync level. The output logic levels are converted to video levels and impedance using three resistors and two diodes (see Figure 2). Including the diodes made it easier to figure out the values of the resistors.

Applications

After the sync generation and pixel blasting is done, the Mega163 still has some cycles leftover. On lines that don't display the image (lines 231 to 262), the CPU is used for about 7 µs for sync generation; the rest of the 63 µs is available for general computation. The number of cycles available is about 55 x 20 x 8 = 8800 (microseconds per line x number of lines x 8 cycles per microsecond). I wrote a few simple applications to see what would fit into 8800 cycles and not interfere with the image generation.

To test the image generation, I wrote a command line interpreter that received commands via the UART from Microsoft HyperTerminal. I implemented commands for line, point, and text. Also, the entire image could be erased (see Photo 1).

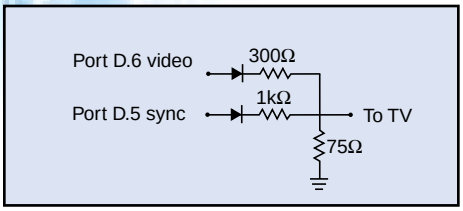

Figure 2: During a 0-V sync pulse, both outputs are low. At the black level, the sync output is high. At the white level, the video output is high.

To avoid image degradation from asynchronous UART events, the UART was polled 60 times per second for an incoming character. The command string was built up until a carriage return occurred, and then interpreted during the interval when no image lines were actually being displayed. This program is a prototype for a student lab in which one CPU generates video (the graphics controller) while another connected CPU computes a game, sends graphics commands, and produces sound effects.

I wrote a diffusion-limited aggregation (DLA) program to check the particle dynamics and the image RAM read-back (see Photo 2). A DLA is a structure that grows by sticking new particles to an existing clump. The DLA generally develops a fractal shape as more particles stick to it. In my code, the DLA starts as a single pixel in the center of the screen and grows every time a randomly moving particle happens to hit it. When a particle hits, it is frozen in place, and a new particle appears at the edge of the screen to diffuse. A single particle can easily move 60 random steps per second, including: erasing the old position; computing two random steps (x and y); drawing the new position; checking for adjacent frozen particles; releasing a new particle (if necessary); and updating the clock display once per second. This program is a prototype for a game-type student lab.

To test its performance, I tried to see how many new characters or lines I could draw before the image generation code overran into the image refresh time. I could write four large characters (4 x 35 = 140 pixels written), 40 small characters, or four lines, each one-half the screen width. A complex image can be built up over several frame times, but any animated pieces of the image must be limited to less than about 140 pixels per frame. Recoding the point routine in assembler speeds up line drawing by about a factor of two; I plan to assign this task as a student lab exercise.

Listing 1— Amusingly enough, the logic to generate 5µs sync pulses exactly fits in about 40 cycles (5 µs); thus the C code is sufficient except for a few lines of code written in assembler.

```
//The sync generator must be entered from Sleep mode to get accurate
timing of the sync pulses. At 8 MHz, all of the sync logic fits in
the 5- s sync pulse.

syncON is initialized to zero
syncOFF is initialized to pull bit 5 high: 0b00100000

The tokens "begin" and "end" are used instead of the usual
C curly brackets
*/
interrupt [TIM1_COMPA] void t1_cmpA(void)
begin

  PORTD = syncON;     //Start the sync pulse
  LineCount ++ ;      //Update the curent scanline number

//Begin inverted (vertical) sync after line 247. Inverting sync means
reversing the values of syncON and syncOFF.
  if (LineCount==248)
  begin
    syncON = 0b00100000;
    syncOFF = 0;
  end

//Back to regular sync after line 250
  if (LineCount==251)
  begin
    syncON = 0;
    syncOFF = 0b00100000;
  end

//Start new frame after line 262
  if (LineCount==263) LineCount = 1;

  PORTD = syncOFF; //End sync pulse
end                //ISR
```

In the Lab

I plan to use this software in an upcoming semester. The performance is good enough for simple games (e.g., Pong or Snake), a clock, a digital voltmeter, or an oscilloscope display. The real-time control of the TV in itself makes a useful learning exercise. I look forward to seeing what kinds of things creative students will do with it. You may download the source code from the Circuit Cellar ftp site or the Cornell web site (instruct1.cit.cornell.edu/courses/ee476/video/index.html). ■

To download the code, go to ftp.circuitcellar.com/pub/Circuit_ Cellar/2003/150/.

References

[1] P. Stang, "TV Paint," Coursework from Stanford University, EE281, Laboratory Assignment no. 4, Handout no. 7 October 2002 www.stanford.edu/class/ee281/handouts/lab4.pdf.

[2] A. Ricci Bitti, "Video DVM," www.geocities.com/CapeCanaveral/Launchpad/3632/dvm.htm.

[3] G. Williamson, "Television," www.williamson-labs.com/480_tv.htm.

[4] R. Gunée, "Software Generated Video," www.efd.lth.se/%7Ee96rg/mc/mc.html#softvideo.

[5] D. Rodgers, Procedural Elements of Computer Graphics, 2nd edition, McGraw-Hill, New York, NY, October 1997.