GCC VIDEO CODE FOR ATMEGA644 MICROCONTROLLER

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical)

by

Shane J Pryor

Project Advisor: Bruce R Land

Degree Date: May 2009

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: GCC Video Code for ATmega644 Microcontroller

Author: Shane J Pryor

Abstract: This project investigates designing C code for use on an Atmel ATmega644 microcontroller. The code outputs an NTSC video signal for display on a television. The project consists of taking a working implementation of the code for a different compiler and microcontroller and porting it to new hardware and software. In addition, the code is changed to be more efficient with a GCC compiler by using fewer hardware registers and taking fewer cycles to execute the same functionality. The resolution of the screen is more than doubled because of the additional on-chip memory of the new hardware. The result of this project is black and white video output with a screen buffer for storing the current output frame. The buffer is manipulated through methods provided in the code on a per-frame basis. Several design choices are presented, with justification for the design methods chosen based upon different trade-offs. The results of the project are presented with a comparison to the previous version and a comparison to the initial design specifications. The project is designed for use by students in a senior level design course in electrical and computer engineering as the foundation for a two week lab to design a video game.

Report Approved by

Project Advisor: _____ Date: _____

Executive Summary

The overall goal of this project was to take video code written for the Atmel ATmega32 microcontroller in the CodeVision AVR environment, and convert the code to run on the ATmega644 microcontroller on the AVR Studio 4 environment using GCC. This project consisted of a few major steps. First, the original code needed to be written for the GCC compiler. The main points of conversion are syntax and usage of code in both the C portion and the assembly portion. Different compilers interpret code in different ways and this had to be accommodated. During the conversion process, there were also parts of the code that were rewritten to be either be more efficient, including improved register use, or more user-friendly to students than the previous version.

Once the GCC version of the code was completed, the code was then converted to run on the ATmega644. Each microcontroller has some difference in features available. Most of the features that have the same functionality between microcontrollers have some small differences in the design of the chip (such as register addresses of associated hardware registers). All of these differences had to be discovered and accounted for through reading of documentation for both chips.

Finally, the last improvement upon the design was increasing the horizontal and vertical resolution of the screen using the additional memory of the ATmega644 compared to the ATmega32. The vertical resolution was increased by 100% and the horizontal resolution was increased by 12.5%. The choice of resolution was based upon keeping the screen proportioned fairly well. There were different choices to outputting the video signal. The selected method was to output through two pins of one of the ports on the microcontroller. These two signals were combined and input to a black and white television through component video input in the NTSC video format.

To complete this project, several pieces of hardware and software had to be learned. To program the ATmega32 used during the process of converting to GCC, an AVR Dragon board was used. The Dragon is not capable of programming an ATmega644, so an STK500 board was used for this. Both were programmed using the ISP interface. In addition to the hardware, the GCC library had to be learned to be able to implement the correct functionality.

Design Problem

Problem:

The goal of this project is to design software running on an Atmel ATmega644 microcontroller to generate a video signal in the black and white NTSC format for input on a component cable to a black and white television. The code needs to be able to output each line of each frame in the timing specifications given by the format. It must also store a buffer of the screen contents in memory. Access to this memory must be fast enough so that output to the port of the microcontroller is not delayed by memory access latency. Also, the buffer must be able to be updated on a per-frame basis. These updates will come from some type of calculations while output is not occurring; therefore, there must also be enough non-output time to do whatever calculations will be required by the application using the video generation code. Finally, there must be some helper functions to provide an interface for manipulating the screen buffer, so someone who uses the code does not have to directly do it themselves.

Previous Work:

A working version of the code has been previously written for the ATmega32 in the CodeVision AVR programming environment by Bruce Land. This code has been used as the basis for a lab in ECE 4760 where the students create a version of the old video game *Lunar Lander*. The code provides functions to the students to interact with the screen buffer so that they do not have to learn exactly how the code works in the short period for the lab. The students must write the computations and use these methods to produce a playable video game. Playable in this sense means the game is winnable and the code the students write does not introduce video artifacts on the screen.

The computations required for this lab involve polling an analog to digital converter (ADC) input, and using the input to control the thrust of their landing vehicle on the screen. Using this acceleration, the new velocity and position of the ship must be calculated. Then, the old position of the ship must be erased from the buffer and the new position of the ship must be drawn to the buffer. The code allows this kind of updating on a per-frame basis to allow for a smooth continuous flight of the ship. There are

several terminating conditions to the game that must also be checked on each movement of the ship: crashing into the side or top of the screen, crashing into the landscape if velocity is too high or ship is angled wrong, or winning conditions. The code must allow all of these computations while still outputting the signal.

For spring 2009, Professor Land had planned to transition ECE 4760 to use the ATmega644 with the AVR Studio 4 (GCC) programming environment. To be able to do this, all of the code used for the labs in the class needed to be converted to run in this new software environment with the new hardware. The lab that uses the most software is the video generation lab. Also, the video generation lab requires precise timing to generate a video signal that looks good on the screen.

Requirements:

The requirements for this project were set at the beginning of the academic year and have not evolved much since. These included making the code faster, making the output buffer a larger resolution, and better register usage. The main questions were how to best implement the requirements on the selected hardware / software combination. A brief summary of the overall requirements is as follows:

- Meet NTSC video format for black and white analog television sets
- Store a buffer to output to the screen each frame at a higher resolution
- Allow fast manipulation of screen buffer through user-friendly methods
- Provide methods to perform fixed point arithmetic to save time / memory over floating point operations

These goals are broad and general and require several stages to complete each. The details of each requirement are described more in detail in the rest of the report; however, it is worth noting here that each of these portions differs in implementation in some level of detail from the previous version of the code. This is due to several different factors, including a different compiler, different features available on the hardware, and different algorithms.

Range of Solutions / Solution Choices

Overall Approach:

The approach to this project was to take the available code and use that as a base to go forward. The first logical step was to take the code and convert it to compile using the GNU Compiler Collection (GCC) compiler for C code running on Atmel microcontrollers. Afterwards, the methods that were changed for optimization were rewritten. While this portion of the project could have occurred at any of the stages, it seemed most appropriate to perform this portion at this stage. It seemed somewhat pointless to port over code to new hardware that was not the final code to be used. If the original code had been ported, the new algorithms would have had to have been written on the new hardware anyway, so this saved some time in the long run. Finally, all of the work associated with the new hardware was done last. The overall approach was fairly straightforward with few design choices.

Video Generation:

The analog video signal had to be produced in the NTSC video format to be able to be used on any standard television in North America. However, there are several design choices that went into this portion of the project. The output of the screen could be in many different formats, so the decision had to be made what the most appropriate output would be. As a brief summary, basically the microcontroller needs to be able to generate a synch signal to synchronize each line and frame, and also a video signal that contains the contents of the screen buffer to place on the screen. The synch signal has to occur at regular intervals to time each line, but the video data can be output as long as there is still time remaining on the line before the next horizontal sync.

This presents a tradeoff between height and width considerations since the new hardware has double the RAM to store the video buffer. The data can be output to the screen as fast as the microcontroller can move each individual bit to the output port. However, the mega644 has a USART that is double-buffered to provide serial output. A possibility for output was to output using the USART instead; when properly configured it could output somewhat faster than setting the port manually. Another method of

increasing the output speed was simply to increase the frequency of the clock used. The maximum frequency that the mega644 can run is 20 MHz. This would provide 25% faster output than the 16 MHz clock generally used with the ATmega microcontrollers in the digital lab.

However, after consideration, neither of these techniques was used. This is because the screen size was set at 200 lines in the vertical direction and 144 individual values in the horizontal direction. Outputting the 144 values using the 16 MHz clock through PORT D provided an image that fit nicely from one side of the screen to the other. If the output speed was increased, then the image would start to be too narrow. It would be useful to output faster if the required effect was a widescreen output for something like a 16:9 aspect ratio. In this case, the screen buffer could be altered to be shorter and wider.

The size of the screen buffer needed to be chosen because the ATmega644 has double the RAM of the ATmega32; therefore, the screen buffer can increase in size once the code runs on the 644. The original size of the buffer was 100 vertical lines by 128 horizontal values. The 100 vertical lines would be displayed twice each – making it appear twice as high as the data actually was; therefore, doubling the number of vertical lines seemed like an intuitive choice for the new hardware. In addition, there was room to expand the screen by the extra 16 values (2 bytes), so this decision was also made. It certainly was not necessary to do this, but there was still enough leftover memory that the screen could be widened with enough free RAM remaining for the students to implement their video game in ECE 4760. Making this change was dependant on modified logic.

Methods:

Most of the design in the other methods came in the choice on whether to write the methods in assembly or C. This design choice was made as the project went along, as it turned out that the GCC compiler was efficient enough that writing the code in C was not much less efficient than writing it in assembler. In addition, most methods performed some calculations based upon input then manipulated the screen buffer; therefore, they needed to be as efficient as possible to save cycles.

Design Documentation

Overview:

The next several pages will contain the steps to completing the project and the implementation details of each step. The first topic discussed will be setting up the system at each stage of the design. This is followed by a description of the hardware digital to analog converter (DAC) built to produce the voltage levels required for NTSC. Next, a description of how the microcontroller can be used to produce an NTSC signal will be covered. Finally, the remaining software methods for editing the video memory and performing fixed point arithmetic are discussed.

Setting up the Hardware and Software:

For this project, a few couple-hundred page manuals had to be read – including those for the AVR GCC library, the mega32, mega644, and a guide provided by Atmel with helpful hints for upgrading / transitioning between their devices. Most of this documentation had to be read in a fair level of detail because there were a lot of nuances to learn about why a particular line of code did not perform the way expected or why a hardware device was not programming.

The first thing I did to start this project was download the software environment and compiler that I would be using. The software environment was AVR Studio 4 with the WinAVR GCC compiler. This compiler is the GCC compiler meant for Atmel microcontrollers. It provides libraries that define all of the hardware registers, interrupts, etc. for each different microcontroller. The compiler syntax and libraries are pretty easy and intuitive to use once the documentation has been read.

From taking ECE 476 in spring 2008, I had a pretty decent working knowledge of how the various features of the mega32 worked, as well as a good understanding of how to write C code for it in the CodeVision AVR. For the final project for this class, my partner and I built a small protoboard to hold a mega32 microcontroller, port connections available to solder wires, programming pins, 16 MHz crystal, and LED hooked to one port pin for test purposes.[1] The programming pins used the six pin ISP programming

interface. This is a standard programming interface for the Atmel microcontrollers. It is utilized on several, if not all, programming boards they produce.

A cheap, easy-to-use programmer that Atmel recently introduced is the AVR Dragon. The Dragon is capable of programming through a few different programming interfaces, including the six pin ISP. After downloading and installing the software, I needed to set-up my project settings to specify my hardware choices. The Dragon requires a USB port to be powered and program another chip. Once everything was configured, I could see the mega32 through the AVR Studio interface; however, I was unable to program the chip. After a good bit of debugging, I discovered that the programming pin that required a connection to Vcc was not being supplied the required voltage from the Dragon. Running a wire from Vcc supply on the protoboard to the programming pin instantly solved the problem.

Once the project was ready to be converted to the mega644, a different programmer needed to be used because the Dragon is not capable of programming the mega644. For this task, an STK500 development board was used. This board requires a separate power supply and a serial cable interface to the PC; however, it is capable of programming many more chips than the Dragon and has other additional features. This switch took a little while to get everything working correctly as well. The board has several jumpers that need to be correct to obtain the correct settings. This includes the jumper to use an external clock source, rather than the internal one of the mega644. The mega644 I was using was brand-new, meaning the fuses of the chip were set to the factory standard. These also needed to be changed to use the external clock source, and the ISP programming interface on the STK500. Finally, the STK500 I was using was also brand-new, so it required a firmware upgrade to be brought up to date.

DAC for Video Output:

A DAC is required to convert the output voltage of the microcontroller to the three voltage levels required by the synch, black color, and white color signals to the television. A schematic of this circuit is provided in Appendix B. This circuit is a simple voltage divider circuit. The sync signal should be ~0V, the black signal should be ~0.3V, and the white signal should be ~1.0V. The voltages are close to these values with the

resistors chosen to be 75 ohms to ground, 330 ohms from video output, and 1000 ohms from sync output. Voltage from the outputs is 5.0V.

Video Generation with Memory:

Video generation follows the NTSC analog television format with a few small changes. An NTSC frame has 525 horizontal lines with two fields of 262.5 lines that are interlaced at 60 Hz each. In this version of the code, one field 262 lines is output per frame at 60 Hz. During these 262 lines, 200 have video output from video memory with the associated sync signal, and the other 62 just have the sync signal for each line (termed the vertical blanking interval). At the appropriate point, sync is inverted to signal it is the end of a frame. The current line number is kept so that each part of the functionality can be performed at the correct time. Also, this allows each line to be output based upon the line count. Each line is stored as 18 bytes of memory, for a total of 3600 bytes of memory used for video.

The output of the signal is very time critical. Each line must start at very regularly spaced intervals or the picture will be severely affected. The mega644 has internal hardware counters that can count time to the resolution of the clock speed. For this application, the amount of time between beginning successive lines is defined by the NTSC standard. With a 16 MHz clock, the amount of time for a line is equal to 1018 clock cycles since 1018 cycles per line multiplied by 262 lines per frame multiplied by 60 frames per second is very close to 16 MHz. To have the microcontroller perform these functions every 1018 cycles, one of the hardware timers is set to enter an interrupt service routine (ISR) every time this number of cycles is reached. Since the microcontroller will finish the current instruction before moving to the ISR, the main routine from which the ISR is entered should be in sleep mode so that the ISR is entered uniformly each time. If the entry is not uniform, each line will begin at a slightly different location on the screen which makes the output jitter horizontally around 1% of where it should be.

Once the interrupt is entered in uniform time, the actual functions of the ISR can be written. For each line, the line count is updated. If the line count reaches 263, it is reset to zero. The sync output is stored in two variables, one for the sync at the beginning of the interrupt and one for right before video output starts. The output port for sync is

assigned these values at the appropriate locations. The sync values stay the same, except for the lines where the inverted sync is required. For the vertical blanking period, this is all the interrupt does; however, if it is a line number where video output occurs, then a few more steps need to be taken. First, a pointer to the location in memory in the screen array where output will start for this line is computed. Then, a nine microsecond delay is introduced. This is because the NTSC signal actually has space built in that is off the screen – this must be accounted for so that video output does not start before the edge of the screen. Finally, each byte is sent out the video port from memory.

Video data is brought from memory to output in a few stages. The compiler recognizes a few pairs of registers as 16 bit address pointers. One such pair is registers 26 and 27. The address where the data starts is loaded into these two registers. Then, a data load of the first byte is performed to register four and the address pointer is incremented by one. A macro is called to output this byte one bit at a time through the port pin. The macro works by taking each successive bit and putting it in a register that is then designated to be output to the address for port D. This procedure is repeated for each of the 18 bytes to be output.

There are a couple things to note about this version of the code against the original version. First, the ISR was changed to use fewer statements and fewer registers. This is a result of the realization that one of the assembler commands was redundant for each bit and only needed to be performed once. With the additional computation cycles, each byte could successively be loaded into the same register instead of loading all the bytes into separate registers initially. Second, the macro to output each bit contains a nop for each bit. The reason for this is so output for each pixel is widened some so a line will fill the entire width of the screen nicely. Originally, each bit had two nops because the screen width had been 16 bytes instead of the 18 it has now.

Buffer Manipulation:

Once there is a buffer for video memory, and a method to output to the television, the user needs a way to edit the memory to place their contents in it. There are a few helper methods written for this purpose. These methods already existed in the original version of the video code; however, several of them were written in assembler. This was

to decrease the number of instructions needed to perform them since the CodeVision compiler was not extremely efficient. It was determined that the optimization on the GCC compiler is good enough so that writing these methods in C instead of assembler is acceptable. C code is preferable in this context because it is easier to understand for someone who is not familiar with how the code works. It is also much easier to edit if additional / different functionality is required in the future. The user interface for the video methods is described in detail in Appendix A.

The logical starting point for this code is a method that alters one pixel in the buffer at a time. This method is video_pt. Since the video memory is stored as an array of bytes, each byte in the memory contains eight horizontal pixels. To access one pixel, first the correct byte must be accessed, and then within this byte one bit is altered. To compute the correct byte index, the given (x, y) pixel coordinates are converted. Since each line of video output has 18 bytes, the vertical (y) value is effectively multiplied by 18 to get the correct line. Then, the horizontal value (x) is effectively divided by 8 and summed with the correct line to get the correct byte. This byte is then set using the appropriate mask for the pixel location. The masks are a set of eight bytes, where each mask has one bit high and the rest low. For instance, if the user wants to change the first pixel to be on, the byte in the video buffer is set to be the logical OR of the byte and the first mask. This ensures that the first pixel is white, while the rest remain the same.

Two additional methods can be implemented based upon what was just discussed. First, a line drawing method can be implemented by using several calls to this point method. Based upon the slope of the line, and it being quantized into pixels, these pixels can be stepped through and drawn with the point method. The algorithm is taken from *Procedural Elements of Computer Graphics* by David Rodgers. Second, a method can be constructed to tell the user what the current value of the video memory is for a particular pixel. The byte containing the pixel can be indexed in the same fashion as in the point drawing method. Then, the individual bit value can be masked as in the point method and the resulting value is returned.

The final set of methods provided deal with outputting characters to the screen. There are two sets of characters that can be output, a 3x5 pixel (small) set and a 5x7 pixel (large) set. For each of these sets, the user can either output a single character at a

provided location or a string of characters given a reference pointer. For small characters, only the Arabic numerals, English capital letters, colon, equals, and space are provided. The large character set has all 128 ASCII characters defined. These character sets were already defined in the code that was being worked with. [2] In this same fashion, any block of pixels defining a unique character could be added, as long as there is sufficient program memory. Character sets are stored in program memory because there is not sufficient memory in RAM.

To draw any of these characters, the user provides the location and the index in the respective array that they wish to draw. The location in turn gives the location of the first byte to change based upon the above methodology. Then, for each of seven bytes, the code reads a byte from program memory and calls video_pt for whether each bit in that character map should be turned on or off (using a mask). Each large character is five bits wide, so five calls to video_pt must be made per byte. For small characters, two bitmaps of each character are stored. In turn, the method requires that the user specify a horizontal location that is a multiple of four. While this restricts location, it allows direct manipulation of the screen buffer from the character method, rather than calling video_pt or reproducing the same functionality. This saves cycles and allows more edits per frame. To edit the buffer, each of the five bytes are read out, and then the appropriate four bits are changed for each buffer byte by masking the input horizontal location to determine whether the first set should be used or the second.

Fixed Point:

A fixed point arithmetic system was implemented to allow users to prevent using floating point notation. While there are intrinsic tradeoffs between the two arithmetic systems in speed, accuracy, and range of numbers that can be represented, speed is really the only important factor in this setting. With this in mind, there are two macros defined to convert between integer and fixed point numbers and two macros to convert between floating point and fixed point numbers. These conversions are simply a shift, a multiply, etc. as the appropriate case may be. Fixed point numbers were chosen to be 16 bits, where 8 bits represent the whole number portion and 8 bits represent the decimal portion.

This assignment is arbitrary as the decimal place can be moved anywhere along the 16 bits, as long as the conversion macros are updated appropriately.

The addition and subtraction operations in fixed point can be performed the same as an integer operation. The multiplication operation is a little trickier. Since the mega644 is an 8-bit hardware, fixed point multiplication can be thought of as four byte-wise multiplications. First, the least significant bytes are multiplied, then pair-wise least and most significant for each number, then finally the most significant. These are summed with the correct shifting. The middle two bytes are the final result. This routine had already been written for the mega644. [3]

System Test Results / Conclusions

Calculations:

The amount of time that computation can be done between displaying frames is based upon the amount of vertical blanking time. That is, the video code outputs during video lines 30-230 of the 262 available for a field in the NTSC format. Therefore, this leaves the remaining lines free to do computations with the exception of the time to output the synchronization pulse and updating the line count. This time can be estimated as 60 lines x 63.5 uSec / line x 16 cycles / uSec = 60960 cycles. If this limit is exceeded the video output will be late, causing video artifacts to appear. This number of cycles is difficult to judge based upon written code; therefore, a brief summary of how many times individual methods can be called per frame is given as a guide to performance.

- Draw a point: ~550 per frame
- Draw a line, 1 pixel long: ~170 per frame; 10 pixels long: ~35 total lines per frame
- Small characters, individually: ~315 per frame; in 10 character strings: ~270 total characters per frame
- Large characters, individually: ~12 per frame; a string of ~14 per frame

These numbers direct relate to how many calculations are performed and how many times values in program memory must be fetched. For drawing a point, the byte index to be altered in the buffer is computed based upon the given (x,y) coordinates. To draw a line, a line drawing algorithm is implemented which calls the video point function multiple times. For each line, there is the same overhead, but the routine to determine each point takes more time than the overhead cost. Therefore, more total pixels can be altered per frame when shorter line segments are drawn.

In the case of characters, they need to fetch the set of bytes from program memory. Obviously larger characters must fetch more than the small characters. In addition, the appropriate manipulations to the screen buffer must be made. For instance,

for every small character, 5 manipulations of the buffer are made directly, and for large characters 35 calls to draw a point are made.

These numbers were determined by trial and error on how many times the particular method could be called before visible artifacts appeared on the screen. This generally would take the form of the top left corner of the screen beginning to warp to the right. This makes sense because the timing constraints for starting the first few lines were not being met, so they were starting late. Eventually, if they started too late, the signal would turn to something similar to noise.

These numbers show that the performance of the code is faster, or in some cases equivalent to, the original version. The original version could draw roughly 500 points in a frame, 500 points worth of lines, 200 3x5 characters, and 16 5x7 characters. [4] The point and small character drawing methods have become more efficient. The number cited for lines is for drawing a lot of long lines in the original version. This number scales back some as the number of function calls increases, as it does in the new version. Slightly fewer large characters can be drawn per frame in this version.

Bouncing Ball:

The first basic test of updating the screen buffer and fixed point calculations came with a simple bouncing ball. The buffer is updated each frame by a call to the video point drawing function to erase the ball, calculating the new position of the ball, and drawing the ball at the new position. Fixed point numbers of x and y position and velocity are stored. There is also a fixed point value stored for gravity and drag. The ball starts with an initial x and y position and velocity. On each frame, the value of velocity is changed in the x direction by drag and in the y direction by drag and gravity. These updates are performed by using the multfix method to calculate the fraction of velocity to subtract in each direction on the current frame. To output the ball, its coordinates must be converted to integer values using the macros.

Pong:

To test the code more fully, a couple video games were created using it. The first was a version of the game pong. The game consists of a boundary that the ball must stay

inside, the ball itself, the paddles the users can control, and a score for each player. The players can move their paddles using two buttons provided to each of them – one of the buttons makes their paddle go up and the other button makes their paddle go down. These buttons are polled on each frame. In addition, the position of the ball is updated on each frame based on a constant velocity. Also, the score of each player is updated if the ball touches the boundary behind either player's paddle. If the ball touches a paddle, it reverses velocity in both directions. The intersecting conditions are tested by a call to video set each frame. All of the objects on the screen are erased and redrawn each frame, with the exception of the boundary box.

Tic-Tac-Toe:

The other game that was designed to test the code was tic-tac-toe. Initially in this game, a grid would be drawn to the middle of the screen that has nine spaces to place an X or O. Above the grid it would print "Player 1's Turn". The player could input their choice of placement by first giving the row and then the column he or she wanted. Then, the choice would appear in the appropriate grid and the new text would read "Player 2's Turn". After each turn, the winning condition of having three marks in a row would be tested for that player. If it was met, the text would change to "Player 1 Wins!", for example.

Conclusions:

The expectations at the beginning of the project were that everything could be implemented correctly and the buffer output methods could operate as fast as the original version of the code. To that end, the goals of performance in this project were met. In addition, the methods written could be called a comparable number of times (in some cases more times) in the allotted time to the original version of the code running in the CodeVision AVR environment. By the beginning of the spring semester, a working implementation of the code had been tested and was ready for use in ECE 4760. This semester, students successfully used the code during the course without problems. For final projects, some student groups are altering the code to do things such as color NTSC, other video games using additional hardware, etc.

This project provided a large amount of knowledge about programming for a real-time system that must meet timing constraints. In this context, it also provided a strong foundation for understanding the features available on current microcontrollers, and how these devices can be applied to so many different applications, in this case a video system. In addition, using different programmers and different microcontrollers gave a little bit of exposure to different available methods to work with programming these devices. Additionally, reading the data sheets for these devices provided good experience being able to read about the functionality of the device and then actually be able to go and implement it. Overall, the project was a success, with few required design changes along the way, and with a wealth of knowledge gained.

References

1. http://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2008/sjp45_rxy2/sjp45_rxy2/index.html

2. http://instruct1.cit.cornell.edu/courses/ee476/video/video2008/VidFixEx2.c

3. http://instruct1.cit.cornell.edu/courses/ee476/labs/s2009/lab4code/multASM.S

4. http://instruct1.cit.cornell.edu/courses/ee476/video/indexCodeVision.html

Appendix A: Software Manual

The following is a brief summary of the functionality provided to alter video memory and perform fixed point arithmetic and conversions:

void video_pt(char x, char y, char c):

Draws a point at pixel position (x, y) with value equal to c, where:

$\{0 \leq x \leq 143, 0 \leq y \leq 199\}$ and c = {0 for black, 1 for white, 2 for invert}

void video_line(char x1, char y1, char x2, char y2, char c):

Draws a line at pixel position (x1, y1) to (x2, y2) with value equal to c, where:

$\{0 \leq x_{1,2} \leq 143, 0 \leq y_{1,2} \leq 199\}$ and c = {0 for black, 1 for white, 2 for invert}

void put_char(char x, char y, char c):

Draws a 5x7 character c at pixel position (x, y) where:

$\{0 \leq x \leq 143, 0 \leq y \leq 199\}$ and c = {ascii value of desired character}

- note that the location of x and y should not specify that part of the character is drawn outside the screen buffer.

void video_puts(char x, char y, char str):*

Draws a string of 5x7 characters referenced by str at pixel position (x, y) where:

$\{0 \leq x \leq 143, 0 \leq y \leq 199\}$ and str = {memory address of beginning of string}

- note that the location of x and y should not specify that part of the character is drawn outside the screen buffer.

void video_smallchar(char x, char y, char c):

Draws a 3x5 character c at pixel position (x, y) where:

$\{0 \leq x \leq 143, 0 \leq y \leq 199\}$ and c = {number or capital letter in smallbitmap array}

- note that the location of x and y should not specify that part of the character is drawn outside the screen buffer and x must be divisible by 4.

void video_putsmalls(char x, char y, char str):*

Draws a string of 3x5 characters referenced by str at pixel position (x, y) where:
$\{0 \leq x \leq 143, 0 \leq y \leq 199\}$ and str = {memory address of beginning of string}
- note that the location of x and y should not specify that part of the character is drawn outside the screen buffer and x must be divisible by 4.

char video_set(char x, char y):

Returns the value of the video memory at pixel position (x, y), where:
$\{0 \leq x \leq 143, 0 \leq y \leq 199\}$

int multfix(int a, int b):

Returns the 16 bit result of multiplying two 16 bit fixed point numbers.

#define int2fix(a):

Converts an 8 bit character whole number to a 16 bit fixed point number.

#define fix2int(a):

Converts a 16 bit fixed point number to an 8 bit integer.

#define float2fix(a):

Converts a single precision floating point number to a 16 bit fixed point number.

#define fix2float(a):

Converts a 16 bit fixed point number to a single precision floating point number.

Appendix B: Pictures / Screenshots

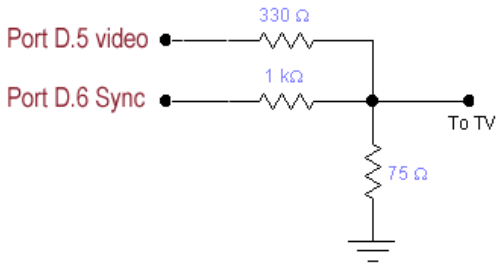

Figure 1: Video DAC

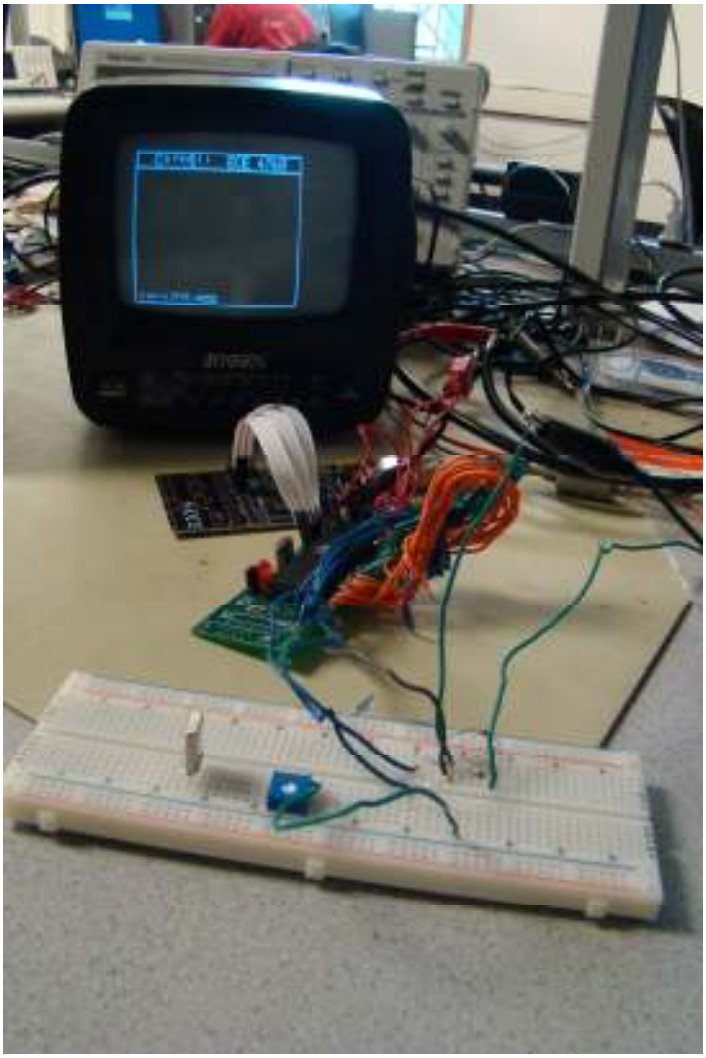

Figure 2: GCC video code running on ATmega32 with AVR Dragon programmer.

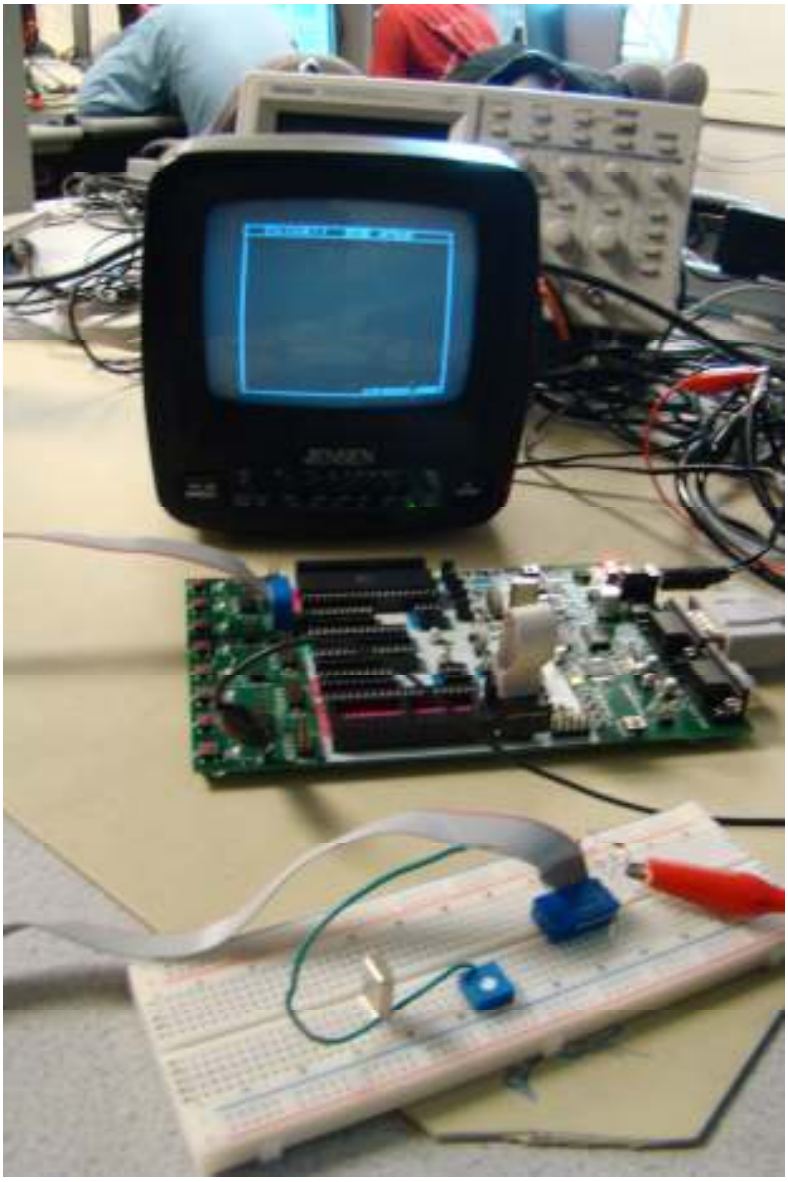

Figure 3: GCC video code running on ATmega644 with STK500 programmer.

Figure 4: Close-up of screen with ATmega32 code running.

Figure 5: Close-up of screen with ATmega644 code running.

Appendix C: C Code

```
// Black and white NTSC video generation with fixed point animation
// D.6 is sync: 330 ohm + diode to 75 ohm resistor
// D.5 is video: 1000 ohm + diode to 75 ohm resistor

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include <stdlib.h>
#include <math.h>
#include <util/delay.h>
#include <avr/sleep.h>

// identify the assembler routine to C
extern int multfix(int a, int b);

//cycles = 63.625 * 16 Note NTSC is 63.55
//but this line duration makes each frame exactly 1/60 sec
//which is nice for keeping a real-time clock
#define lineTime 1018
#define ScreenTop 30
#define ScreenBot 230

//sync
volatile char syncON, syncOFF;

//line number in the current frame
volatile int LineCount;

//200v x 144h - screen buffer and pointer
char screen[3600];
int* screenindex;

//One bit masks
char pos[8] = {0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};

//================================
//3x5 font numbers, then letters
//packed two per definition for fast
//copy to the screen at x-position divisible by 4
prog_char smallbitmap[39][5] = {
       //0
        {0b11101110,
        0b10101010,
        0b10101010,
        0b10101010,
        0b11101110},
        //1
        {0b01000100,
        0b11001100,
        0b01000100,
        0b01000100,
        0b11101110},
        //2
        {0b11101110,
```

```
0b00100010,
0b11101110,
0b10001000,
0b11101110},
//3
{0b11101110,
0b00100010,
0b11101110,
0b00100010,
0b11101110},
//4
{0b10101010,
0b10101010,
0b11101110,
0b00100010,
0b00100010},
//5
{0b11101110,
0b10001000,
0b11101110,
0b00100010,
0b11101110},
//6
{0b11001100,
0b10001000,
0b11101110,
0b10101010,
0b11101110},
//7
{0b11101110,
0b00100010,
0b01000100,
0b10001000,
0b10001000},
//8
{0b11101110,
0b10101010,
0b11101110,
0b10101010,
0b11101110},
//9
{0b11101110,
0b10101010,
0b11101110,
0b00100010,
0b01100110},
//:
{0b00000000,
0b01000100,
0b00000000,
0b01000100,
0b00000000},
//=
{0b00000000,
0b11101110,
0b00000000,
0b11101110,
```

```
0b00000000},
//blank
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//A
{0b11101110,
0b10101010,
0b11101110,
0b10101010,
0b10101010},
//B
{0b11001100,
0b10101010,
0b11101110,
0b10101010,
0b11001100},
//C
{0b11101110,
0b10001000,
0b10001000,
0b10001000,
0b11101110},
//D
{0b11001100,
0b10101010,
0b10101010,
0b10101010,
0b11001100},
//E
{0b11101110,
0b10001000,
0b11101110,
0b10001000,
0b11101110},
//F
{0b11101110,
0b10001000,
0b11101110,
0b10001000,
0b10001000},
//G
{0b11101110,
0b10001000,
0b10001000,
0b10101010,
0b11101110},
//H
{0b10101010,
0b10101010,
0b11101110,
0b10101010,
0b10101010},
//I
{0b11101110,
```

```
0b01000100,
0b01000100,
0b01000100,
0b11101110},
//J
{0b00100010,
0b00100010,
0b00100010,
0b10101010,
0b11101110},
//K
{0b10001000,
0b10101010,
0b11001100,
0b11001100,
0b10101010},
//L
{0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b11101110},
//M
{0b10101010,
0b11101110,
0b11101110,
0b10101010,
0b10101010},
//N
{0b00000000,
0b11001100,
0b10101010,
0b10101010,
0b10101010},
//O
{0b01000100,
0b10101010,
0b10101010,
0b10101010,
0b01000100},
//P
{0b11101110,
0b10101010,
0b11101110,
0b10001000,
0b10001000},
//Q
{0b01000100,
0b10101010,
0b10101010,
0b11101110,
0b01100110},
//R
{0b11101110,
0b10101010,
0b11001100,
0b11101110,
```

```
        0b10101010},
        //S
        {0b11101110,
        0b10001000,
        0b11101110,
        0b00100010,
        0b11101110},
        //T
        {0b11101110,
        0b01000100,
        0b01000100,
        0b01000100,
        0b01000100},
        //U
        {0b10101010,
        0b10101010,
        0b10101010,
        0b10101010,
        0b11101110},
        //V
        {0b10101010,
        0b10101010,
        0b10101010,
        0b10101010,
        0b01000100},
        //W
        {0b10101010,
        0b10101010,
        0b11101110,
        0b11101110,
        0b10101010},
        //X
        {0b00000000,
        0b10101010,
        0b01000100,
        0b01000100,
        0b10101010},
        //Y
        {0b10101010,
        0b10101010,
        0b01000100,
        0b01000100,
        0b01000100},
        //Z
        {0b11101110,
        0b00100010,
        0b01000100,
        0b10001000,
        0b11101110}
};

//================================================
// Full ascii 5x7 char set
// Designed by: David Perez de la Cruz and Ed Lau
// see:
http://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2005/dp93
/index.html
```

```
prog_char ascii[128][7] = {
        //0
        {0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000},
        //1
        {0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000},
        //2
        {0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000},
        //3
        {0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000},
        //4
        {0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000},
        //5
        {0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000},
        //6
        {0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
```

```
0b00000000},
//7
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//8
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//9
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//10
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//11
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//12
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//13
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
```

```
//14
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//15
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//16
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//17
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//18
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//19
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//20
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//21
```

```
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//22
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//23
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//24
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//25
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//26
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//27
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//28
{0b00000000,
```

```
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//29
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//30
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//31
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//32 Space
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//33 Exclamation !
{0b01100000,
0b01100000,
0b01100000,
0b01100000,
0b00000000,
0b00000000,
0b01100000},
//34 Quotes "
{0b01010000,
0b01010000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//35 Number #
{0b00000000,
0b01010000,
```

```
0b11111000,
0b01010000,
0b11111000,
0b01010000,
0b00000000},
//36 Dollars $
{0b01110000,
0b10100000,
0b10100000,
0b01110000,
0b00101000,
0b00101000,
0b01110000},
//37 Percent %
{0b01000000,
0b10101000,
0b01010000,
0b00100000,
0b01010000,
0b10101000,
0b00010000},
//38 Ampersand &
{0b00100000,
0b01010000,
0b10100000,
0b01000000,
0b10101000,
0b10010000,
0b01101000},
//39 Single Quote '
{0b01000000,
0b01000000,
0b01000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//40 Left Parenthesis (
{0b00010000,
0b00100000,
0b01000000,
0b01000000,
0b01000000,
0b00100000,
0b00010000},
//41 Right Parenthesis )
{0b01000000,
0b00100000,
0b00010000,
0b00010000,
0b00010000,
0b00100000,
0b01000000},
//42 Star *
{0b00010000,
0b00111000,
0b00010000,
```

```
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//43 Plus +
{0b00000000,
0b00100000,
0b00100000,
0b11111000,
0b00100000,
0b00100000,
0b00000000},
//44 Comma ,
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00010000,
0b00010000},
//45 Minus -
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b11111000,
0b00000000,
0b00000000},
//46 Period .
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00010000},
// 47 Backslash /
{0b00000000,
0b00001000,
0b00010000,
0b00100000,
0b01000000,
0b10000000,
0b00000000},
// 48 Zero
{0b01110000,
0b10001000,
0b10011000,
0b10101000,
0b11001000,
0b10001000,
0b01110000},
//49 One
{0b00100000,
0b01100000,
0b00100000,
0b00100000,
```

```
0b00100000,
0b00100000,
0b01110000},
//50 Two
{0b01110000,
0b10001000,
0b00001000,
0b00010000,
0b00100000,
0b01000000,
0b11111000},
 //51 Three
{0b11111000,
0b00010000,
0b00100000,
0b00010000,
0b00001000,
0b10001000,
0b01110000},
//52 Four
{0b00010000,
0b00110000,
0b01010000,
0b10010000,
0b11111000,
0b00010000,
0b00010000},
//53 Five
{0b11111000,
0b10000000,
0b11110000,
0b00001000,
0b00001000,
0b10001000,
0b01110000},
//54 Six
{0b01000000,
0b10000000,
0b10000000,
0b11110000,
0b10001000,
0b10001000,
0b01110000},
//55 Seven
{0b11111000,
0b00001000,
0b00010000,
0b00100000,
0b01000000,
0b10000000,
0b10000000},
//56 Eight
{0b01110000,
0b10001000,
0b10001000,
0b01110000,
0b10001000,
```

```
0b10001000,
0b01110000},
//57 Nine
{0b01110000,
0b10001000,
0b10001000,
0b01111000,
0b00001000,
0b00001000,
0b00010000},
//58 :
{0b00000000,
0b00000000,
0b00100000,
0b00000000,
0b00000000,
0b00000000,
0b00100000},
//59 ;
{0b00000000,
0b00000000,
0b00100000,
0b00000000,
0b00000000,
0b00100000,
0b00100000},
//60 <
{0b00000000,
0b00011000,
0b01100000,
0b10000000,
0b01100000,
0b00011000,
0b00000000},
//61 =
{0b00000000,
0b00000000,
0b01111000,
0b00000000,
0b01111000,
0b00000000,
0b00000000},
//62 >
{0b00000000,
0b11000000,
0b00110000,
0b00001000,
0b00110000,
0b11000000,
0b00000000},
//63 ?
{0b00110000,
0b01001000,
0b00010000,
0b00100000,
0b00100000,
0b00000000,
```

```
0b00100000},
//64 @
{0b01110000,
0b10001000,
0b10111000,
0b10101000,
0b10010000,
0b10001000,
0b01110000},
//65 A
{0b01110000,
0b10001000,
0b10001000,
0b10001000,
0b11111000,
0b10001000,
0b10001000},
//B
{0b11110000,
0b10001000,
0b10001000,
0b11110000,
0b10001000,
0b10001000,
0b11110000},
//C
{0b01110000,
0b10001000,
0b10000000,
0b10000000,
0b10000000,
0b10001000,
0b01110000},
//D
{0b11110000,
0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b11110000},
//E
{0b11111000,
0b10000000,
0b10000000,
0b11111000,
0b10000000,
0b10000000,
0b11111000},
//F
{0b11111000,
0b10000000,
0b10000000,
0b11111000,
0b10000000,
0b10000000,
0b10000000},
```

```
//G
{0b01110000,
0b10001000,
0b10000000,
0b10011000,
0b10001000,
0b10001000,
0b01110000},
//H
{0b10001000,
0b10001000,
0b10001000,
0b11111000,
0b10001000,
0b10001000,
0b10001000},
//I
{0b01110000,
0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b01110000},
//J
{0b00111000,
0b00010000,
0b00010000,
0b00010000,
0b00010000,
0b10010000,
0b01100000},
//K
{0b10001000,
0b10010000,
0b10100000,
0b11000000,
0b10100000,
0b10010000,
0b10001000},
//L
{0b10000000,
0b10000000,
0b10000000,
0b10000000,
0b10000000,
0b10000000,
0b11111000},
//M
{0b10001000,
0b11011000,
0b10101000,
0b10101000,
0b10001000,
0b10001000,
0b10001000},
//N
```

```
{0b10001000,
0b10001000,
0b11001000,
0b10101000,
0b10011000,
0b10001000,
0b10001000},
//O
{0b01110000,
0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b01110000},
//P
{0b11110000,
0b10001000,
0b10001000,
0b11110000,
0b10000000,
0b10000000,
0b10000000},
//Q
{0b01110000,
0b10001000,
0b10001000,
0b10001000,
0b10101000,
0b10010000,
0b01101000},
//R
{0b11110000,
0b10001000,
0b10001000,
0b11110000,
0b10100000,
0b10010000,
0b10001000},
//S
{0b01111000,
0b10000000,
0b10000000,
0b01110000,
0b00001000,
0b00001000,
0b11110000},
//T
{0b11111000,
0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00100000},
//U
{0b10001000,
```

```
0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b01110000},
//V
{0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b10001000,
0b01010000,
0b00100000},
//W
{0b10001000,
0b10001000,
0b10001000,
0b10101000,
0b10101000,
0b10101000,
0b01010000},
//X
{0b10001000,
0b10001000,
0b01010000,
0b00100000,
0b01010000,
0b10001000,
0b10001000},
//Y
{0b10001000,
0b10001000,
0b10001000,
0b01010000,
0b00100000,
0b00100000,
0b00100000},
//Z
{0b11111000,
0b00001000,
0b00010000,
0b00100000,
0b01000000,
0b10000000,
0b11111000},
//91 [
{0b11100000,
0b10000000,
0b10000000,
0b10000000,
0b10000000,
0b10000000,
0b11100000},
//92 (backslash)
{0b00000000,
0b10000000,
```

```
0b01000000,
0b00100000,
0b00010000,
0b00001000,
0b00000000},
//93 ]
{0b00111000,
0b00001000,
0b00001000,
0b00001000,
0b00001000,
0b00001000,
0b00111000},
//94 ^
{0b00100000,
0b01010000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//95 _
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b11111000},
//96 `
{0b10000000,
0b01000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000},
//97 a
{0b00000000,
0b01100000,
0b00010000,
0b01110000,
0b10010000,
0b01100000,
0b00000000},
//98 b
{0b10000000,
0b10000000,
0b11100000,
0b10010000,
0b10010000,
0b11100000,
0b00000000},
//99 c
{0b00000000,
0b00000000,
0b01110000,
```

```
0b10000000,
0b10000000,
0b01110000,
0b00000000},
// 100 d
{0b00010000,
0b00010000,
0b01110000,
0b10010000,
0b10010000,
0b01110000,
0b00000000},
//101 e
{0b00000000,
0b01100000,
0b10010000,
0b11110000,
0b10000000,
0b01110000,
0b00000000},
//102 f
{0b00110000,
0b01000000,
0b11100000,
0b01000000,
0b01000000,
0b01000000,
0b00000000},
//103 g
{0b00000000,
0b01100000,
0b10010000,
0b01110000,
0b00010000,
0b00010000,
0b01100000},
//104 h
{0b10000000,
0b10000000,
0b11100000,
0b10010000,
0b10010000,
0b10010000,
0b00000000},
//105 i
{0b00000000,
0b00100000,
0b00000000,
0b00100000,
0b00100000,
0b00100000,
0b00000000},
//106 j
{0b00000000,
0b00010000,
0b00000000,
0b00010000,
```

```
0b00010000,
0b00010000,
0b01100000},
//107 k
{0b10000000,
0b10010000,
0b10100000,
0b11000000,
0b10100000,
0b10010000,
0b00000000},
//108 l
{0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00000000},
//109 m
{0b00000000,
0b00000000,
0b01010000,
0b10101000,
0b10101000,
0b10101000,
0b00000000},
//110 n
{0b00000000,
0b00000000,
0b01100000,
0b10010000,
0b10010000,
0b10010000,
0b00000000},
//111 o
{0b00000000,
0b00000000,
0b01100000,
0b10010000,
0b10010000,
0b01100000,
0b00000000},
//112 p
{0b00000000,
0b00000000,
0b01100000,
0b10010000,
0b11110000,
0b10000000,
0b10000000},
//113 q
{0b00000000,
0b00000000,
0b01100000,
0b10010000,
0b11110000,
```

```
0b00010000,
0b00010000},
//114 r
{0b00000000,
0b00000000,
0b10111000,
0b01000000,
0b01000000,
0b01000000,
0b00000000},
//115 s
{0b00000000,
0b00000000,
0b01110000,
0b01000000,
0b00010000,
0b01110000,
0b00000000},
//116 t
{0b01000000,
0b01000000,
0b11100000,
0b01000000,
0b01000000,
0b01000000,
0b00000000},
// 117u
{0b00000000,
0b00000000,
0b10010000,
0b10010000,
0b10010000,
0b01100000,
0b00000000},
//118 v
{0b00000000,
0b00000000,
0b10001000,
0b10001000,
0b01010000,
0b00100000,
0b00000000},
//119 w
{0b00000000,
0b00000000,
0b10101000,
0b10101000,
0b01010000,
0b01010000,
0b00000000},
//120 x
{0b00000000,
0b00000000,
0b10010000,
0b01100000,
0b01100000,
0b10010000,
```

```
0b00000000},
//121 y
{0b00000000,
0b00000000,
0b10010000,
0b10010000,
0b01100000,
0b01000000,
0b10000000},
//122 z
{0b00000000,
0b00000000,
0b11110000,
0b00100000,
0b01000000,
0b11110000,
0b00000000},
//123 {
{0b00100000,
0b01000000,
0b01000000,
0b10000000,
0b01000000,
0b01000000,
0b00100000},
//124 |
{0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00100000,
0b00100000},
//125 }
{0b00100000,
0b00010000,
0b00010000,
0b00001000,
0b00010000,
0b00010000,
0b00100000},
//126 ~
{0b00000000,
0b00000000,
0b01000000,
0b10101000,
0b00010000,
0b00000000,
0b00000000},
//127 DEL
{0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000,
0b00000000}
```

```
};

//macro to put a byte to the screen
asm(".MACRO videobits\n\t"

      "BST  R4,7\n\t"
      "BLD  R30,5\n\t"
      "OUT  0x0b,R30\n\t"
      "NOP \n\t"

      "BST  R4,6\n\t"
      "BLD  R30,5\n\t"
      "OUT  0x0b,R30\n\t"
      "NOP \n\t"

      "BST  R4,5\n\t"
      "BLD  R30,5\n\t"
      "OUT  0x0b,R30\n\t"
      "NOP \n\t"

      "BST  R4,4\n\t"
      "BLD  R30,5\n\t"
      "OUT  0x0b,R30\n\t"
      "NOP \n\t"

      "BST  R4,3\n\t"
      "BLD  R30,5\n\t"
      "OUT  0x0b,R30\n\t"
      "NOP \n\t"

      "BST  R4,2\n\t"
      "BLD  R30,5\n\t"
      "OUT  0x0b,R30\n\t"
      "NOP \n\t"

      "BST  R4,1\n\t"
      "BLD  R30,5\n\t"
      "OUT  0x0b,R30\n\t"
      "NOP \n\t"

      "BST  R4,0\n\t"
      "BLD  R30,5\n\t"
      "OUT  0x0b,R30\n\t"
      "NOP \n\t"

      ".ENDM\n\t"
   );

// puts 18 bytes (1 line) to the screen
void byteblast() {

      asm(  "LDS  R26,screenindex\n\t"
            "LDS  R27,screenindex+1\n\t"

            "LDI R30, 0x40\n\t"

            "LD  R4,X+\n\t"
```

```
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X+\n\t"
"videobits\n\t"

"LD  R4,X\n\t"
"videobits\n\t"

"CLT\n\t"
"LDI R30, 0x40\n\t"

"BLD  R30,5\n\t"
```

```
            "OUT  0x0b,R30\n\t"
         );
}

//==================================
//This is the sync generator and raster generator. It MUST be entered
from
//sleep mode to get accurate timing of the sync pulses

ISR (TIMER1_COMPA_vect) {
      //start the Horizontal sync pulse
      PORTD = syncON;

      //update the current scanline number
      LineCount++;

      //begin inverted (Vertical) synch after line 247
      if (LineCount==248) {
        syncON = 0b01000000;
        syncOFF = 0;
      }

      //back to regular sync after line 250
      if (LineCount==251)     {
        syncON = 0;
        syncOFF = 0b01000000;
      }

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

      //adjust to make 5 us pulses
      _delay_us(2);

      //end sync pulse
      PORTD = syncOFF;

      if (LineCount < ScreenBot && LineCount >= ScreenTop) {
        //compute offset into screen array
        screenindex = (int *)(screen + ((LineCount - ScreenTop) << 4) +
                      ((LineCount - ScreenTop) << 1));

      //center image on screen
      _delay_us(9);

      //blast the data to the screen
      byteblast();
   }
}

//==================================
//plot one point
//at x,y with color 1=white 0=black 2=invert
void video_pt(char x, char y, char c) {
      //each line has 18 bytes
      //calculate i based upon this and x,y
```

```
      // the byte with the pixel in it
      int i = (x >> 3) + ((int)y<<4) + ((int)y<<1);

      if (c==1)
           screen[i] = screen[i] | pos[x & 7];
      else if (c==0)
           screen[i] = screen[i] & ~pos[x & 7];
      else
           screen[i] = screen[i] ^ pos[x & 7];
}

//==================================
//plot a line
//at x1,y1 to x2,y2 with color 1=white 0=black 2=invert
//NOTE: this function requires signed chars
//Code is from David Rodgers,
//"Procedural Elements of Computer Graphics",1985
void video_line(char x1, char y1, char x2, char y2, char c) {
      int e;
      signed int dx,dy,j, temp;
      signed char s1,s2, xchange;
      signed int x,y;

      x = x1;
      y = y1;

      //take absolute value
      if (x2 < x1) {
            dx = x1 - x2;
            s1 = -1;
      }

      else if (x2 == x1) {
            dx = 0;
            s1 = 0;
      }

      else {
            dx = x2 - x1;
            s1 = 1;
      }

      if (y2 < y1) {
            dy = y1 - y2;
            s2 = -1;
      }

      else if (y2 == y1) {
            dy = 0;
            s2 = 0;
      }

      else {
            dy = y2 - y1;
            s2 = 1;
      }
```

```
      xchange = 0;

      if (dy>dx) {
            temp = dx;
            dx = dy;
            dy = temp;
            xchange = 1;
      }

      e = ((int)dy<<1) - dx;

      for (j=0; j<=dx; j++) {
            video_pt(x,y,c);

            if (e>=0) {
                  if (xchange==1) x = x + s1;
                  else y = y + s2;
                  e = e - ((int)dx<<1);
            }

            if (xchange==1) y = y + s2;
            else x = x + s1;

            e = e + ((int)dy<<1);
      }
}

//================================
// put a big character on the screen
// c is index into bitmap
void video_putchar(char x, char y, char c) {
      char i;
      char y_pos;
      uint8_t j;

      for (i=0;i<7;i++) {
        y_pos = y + i;

        j = pgm_read_byte(((uint16_t)(ascii)) + c*7 + i);

        video_pt(x,   y_pos, (j & 0x80)==0x80);
        video_pt(x+1, y_pos, (j & 0x40)==0x40);
        video_pt(x+2, y_pos, (j & 0x20)==0x20);
        video_pt(x+3, y_pos, (j & 0x10)==0x10);
        video_pt(x+4, y_pos, (j & 0x08)==0x08);
    }
}

//================================
// put a string of big characters on the screen
void video_puts(char x, char y, char *str) {
      uint8_t i;
      for (i=0; str[i]!=0; i++) {
            video_putchar(x,y,str[i]);
            x = x+6;
      }
}
```

```
//==================================
// put a small character on the screen
// x-coord must be on divisible by 4
// c is index into bitmap
void video_smallchar(char x, char y, char c) {
      char mask;
      int i=((int)x>>3) + ((int)y<<4) + (((int)y)<<1);

      if (x == (x & 0xf8)) mask = 0x0f;     //f8
      else mask = 0xf0;

      uint8_t j = pgm_read_byte(((uint16_t)(smallbitmap)) + c*5);
      screen[i]    = (screen[i] & mask) | (j & ~mask);

      j = pgm_read_byte(((uint16_t)(smallbitmap)) + c*5 + 1);
      screen[i+18] = (screen[i+18] & mask) | (j & ~mask);

      j = pgm_read_byte(((uint16_t)(smallbitmap)) + c*5 + 2);
      screen[i+36] = (screen[i+36] & mask) | (j & ~mask);

      j = pgm_read_byte(((uint16_t)(smallbitmap)) + c*5 + 3);
      screen[i+54] = (screen[i+54] & mask) | (j & ~mask);

      j = pgm_read_byte(((uint16_t)(smallbitmap)) + c*5 + 4);
      screen[i+72] = (screen[i+72] & mask) | (j & ~mask);
}

//==================================
// put a string of small characters on the screen
// x-cood must be on divisible by 4
void video_putsmalls(char x, char y, char *str) {
      uint8_t i;
      x = x & 0b1111100; //make it divisible by 4
      for (i = 0; str[i] != 0; i++) {
        if (str[i] >= 0x30 && str[i] <= 0x3a)
          video_smallchar(x, y, str[i] - 0x30);

        else video_smallchar(x, y, str[i]-0x40+12);
          x += 4;
      }
}

//==================================
//return the value of one point
//at x,y with color 1=white 0=black 2=invert
char video_set(char x, char y) {
      //The following construction
      //detects exactly one bit at the x,y location
      int i = (x>>3) + ((int)y<<4) + ((int)y<<1);

      return (screen[i] & 1<<(7-(x & 0x7)));
}

//=== fixed conversion macros ========================================
#define int2fix(a)   (((int)(a))<<8)       //Convert char to fix. a is
                                           //a char
```

```
#define fix2int(a)   ((uint8_t)((a)>>8))   //Convert fix to int. a is
                                           //an int
#define float2fix(a) ((int)((a)*256.0))    //Convert float to fix. a is
                                           //a float
#define fix2float(a) (((float)(a))/256.0)  //Convert fix to float. a is
                                           //an int

//=== animation stuff
=================================================
unsigned int x, y;
int vx, vy, g, drag;
char cu1[]="Cornell  ECE 4760";
char cu2[]="COPYRIGHT:2009";


//==================================
// set up the ports and timers
int main() {
  //init timer 1 to generate sync
  OCR1A = lineTime;     //One NTSC line

  TCCR1B = 0x09;  //full speed; clear-on-match
  TCCR1A = 0x00;  //turn off pwm and oc lines
  TIMSK1 = 0x02;  //enable interrupt T1 cmp A

  //init ports
  DDRD = 0xf0;          //video out

  DDRC = 0xff;      //LEDs

  //initialize synch constants
  LineCount = 1;

  syncON = 0b00000000;
  syncOFF = 0b01000000;

  //Print "CORNELL" message
  video_puts(13,2,cu1);

  //Print "Copyright" message
  video_putsmalls(86,192,cu2);

  //side lines
  #define width 142
  #define height 199

  video_line(0,0,0,height,1);
  video_line(width,0,width,height,1);

  video_line(0,10,width,10,1);
  video_line(0,0,width,0,1);
  video_line(0,height,width,height,1);


  //init animation
  // initial position
  x = int2fix(20);
```

```
  y = int2fix(40);

  // initial velocity
  vx = float2fix(1.5);
  vy = float2fix(0.0);

  // gravity and drag
  g = float2fix(0.018);
  drag = float2fix(0.004);

  video_pt(fix2int(x),fix2int(y),2);
  ////////////////////////

  // Set up single video line timing
  sei();
  set_sleep_mode(SLEEP_MODE_IDLE);
  sleep_enable();


  //The following loop executes once/video line during lines
  //1-230, then does all of the frame-end processing
  for(;;) {

    //stall here until next line starts
    //use sleep to make entry into sync ISR uniform time

      sleep_cpu();

      //The following code executes during the vertical blanking
      //Code here can be as long as
      //a total of 60 lines x 63.5 uSec/line x 8 cycles/uSec

      if (LineCount == 231) {

        //animation
        //erase old ball using XOR mode
        video_pt(fix2int(x),fix2int(y),2);

        // new vel = old vel + acceleration*dt (dt=1)
        vy = vy + g - multfix(drag,vy);
        vx = vx - multfix(drag,vx);

        // new pos = old pos + velocity*dt (dt=1)
        x = x + vx;
        y = y + vy;

        // Bounds check -- fixed notation
        if (x>0x8d00) { vx = -vx; x=0x8d00; }    // right edge
        if (x<0x0100) { vx = -vx; x=0x0100; }    // left edge
        if (y>0xc600) { vy = -vy; y=0xc600; }    // bottom edge

        video_pt(fix2int(x),fix2int(y),2);

     } //line 231
  }  //for
}  //main
```

```
;*****************************************************************
;*
;* FUNCTION
;*    muls16x16
;* DECRIPTION
;*    Signed multiply of two 16bits numbers with 16 bits result.
;* USAGE
;*    r25:r24 = r23:r22 * r25:r24
;*****************************************************************

;int multfix(int a, int b)

.global multfix
multfix:
      ;input parameters are in r23:r22(hi:lo) and r25:r24

      ;b already in right place -- 2nd parameter is in r22:23


      mov  r20,r24 ;load a -- first parameter is in r24:25
      mov  r21,r25


      muls r23, r21     ; (signed)ah * (signed)bh
      mov    r25, r0         ;r18, r0"
      mul    r22, r20         ; al * bl"
      mov  r24, r1      ;movw r17:r16, r1:r0"

      mulsu r23, r20    ; (signed)ah * bl
      add    r24, r0         ;r17, r0"
      adc    r25, r1         ;r18, r1"

      mulsu r21, r22    ; (signed)bh * al
      add    r24, r0         ;r17, r0"
      adc    r25, r1         ;r18, r1"

      clr  r1                ; required by GCC


      ;return values are in 25:r24 (hi:lo)

      ret
```