Low Cost, Audio Speed Digital Oscilloscope

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

Morgan D. Jones

Project Advisor: Bruce R. Land

Degree Date: May 2008

Abstract

The goal of the project was to design a low-cost digital oscilloscope targeted at students, hobbyists, and researchers. The requirements of the oscilloscope were three-fold: 1) cost less than $100, 2) capture frequencies over the range 20Hz–20kHz, and 3) constructable with a basic soldering iron and limited soldering skills. The primary parts of the oscilloscope are two Atmel ATmega644 microcontrollers, an Analog Device AD7825 ADC, and an NTSC television. One of the microcontrollers is used to generate an NTSC television signal, and the other is responsible for acquiring and processing samples from the ADC.

The oscilloscope implements many of the same operations as a commercial oscilloscope. The user can start/stop the display, arm a single sequence, adjust the time division, adjust the voltage division, and adjust the trigger value. When the display is stopped, the user can also control a cursor which has time and voltage readouts. The supported time divisions are 0.5ms–5s, and the supported voltage divisions are 156mV–5V.

All the specified requirements were met satisfactorily. The total cost of parts required for the oscilloscope was only $65.51, well within the specified budget. The oscilloscope can display signals up to 40kHz with at least 10 samples per cycle, which is better than the requirement of 20Hz–20kHz. Finally, the oscilloscope was designed entirely with parts that could be soldered by hand.

Report Approved by
Project Advisor: ______________________ Date: ______________

Executive Summary

The goal of the project was to design a low-cost digital oscilloscope targeted at students, hobbyists, and researchers unable to afford a commercial oscilloscope. The requirements of the oscilloscope were three-fold: 1) cost less than $100, 2) capture frequencies over the range 20Hz–20kHz, and 3) constructable with a basic soldering iron and limited soldering skills.

To fulfill the requirements, the oscilloscope would need, at bare minimum, a processing unit, an ADC (Analog-to-Digital Converter), a display device, and a user interface. The best option for the processing unit was determined to be an Atmel AVR microcontroller due to it's low cost, easily soldered package, and availability of high-quality free software development tools. The ADC was narrowed to any stand-alone ADC capable of at least 200kHz sampling rate and available in a package that could be soldered by hand. A television was picked as the only display device that could meet all the requirements. The user interface was specified to be simple push buttons.

Ultimately, two Atmel ATmega644 microcontrollers were used. One (the "video" microcontroller) was used to generate the NTSC television signal, and the other (the "analog" microcontroller) was responsible for acquiring and processing samples from the ADC. The two microcontrollers communicate via an SPI bus with additional hand-shaking lines. The specific ADC chosen was an AD7825 8-bit parallel unit from Analog Devices.

The incoming analog signal is conditioned in a simple analog frontend. The signal is first amplified by a programmable gain amplifier controlled by the analog microcontroller. After amplification, the signal passes through a 1/2 attenuator to adjust for the 2.5V input range of the ADC. The entire analog frontend is referenced to 2.5V virtual ground.

The samples from the ADC are processed by the analog microcontroller. The analog microcontroller then passes the processed samples onto to the video microcontroller. The video microcontroller, in turn, generates the appropriate NTSC video signal to display the samples on the television. The analog microcontroller can also control the header and footer of the screen. The header and footer are strings of text along the top and bottom of the screen.

The oscilloscope implements many of the same operations as a commercial oscilloscope. The user can start/stop the display, arm a single sequence, adjust the time division, adjust the voltage division, and adjust the trigger value. When the display is stopped, the user can also control a cursor which has time and voltage readouts. The supported time divisions are 0.5ms–5s, and the supported voltage divisions are 156mV–5V.

All the specified requirements were met satisfactorily. The total cost of parts required for the oscilloscope was only $65.51, well within the specified budget. The oscilloscope can display signals up to 40kHz with at least 10 samples per cycle, which is better than the requirement of 20Hz–20kHz. Finally, the oscilloscope was designed entirely with parts that could be soldered by hand. Although not trivial to assemble, anyone with at least a little soldering experience should be able to build it.

Contents

1. Requirements **2**

2. Possible Solutions **3**

3. Background **5**
3.1. NTSC Black and White Television Signal 5
3.2. Generating NTSC Video Signal 5

4. Hardware Design **7**
4.1. Power 7
4.2. Microcontrollers 7
4.3. Video DAC 7
4.4. Analog Frontend 9
4.5. ADC 9
4.6. In-circuit Programming 10
4.7. Printed Circuit Board 10

5. Software **12**
5.1. Analog Microcontroller 12
5.1.1. User Interface 12
5.1.2. Managing Sample Rate 13
5.1.3. ADC Conversion 13
5.1.4. Setting Gain 15
5.1.5. Data Transmission to Video Microcontroller 16
5.2. Video Microcontroller 17
5.2.1. Character Look-up Table 17
5.2.2. Graph Raster 17
5.2.3. Scanline Generation 18
5.2.4. Data Reception from Analog Microcontroller 19

6. Results **21**

A. Schematics **23**

B. PCB Layout **27**

C. Bill of Materials **28**

D. Analog Code Listing **29**

E. Video Code Listing **47**

1. Requirements

Professional quality digital oscilloscopes can cost a thousand dollars or more. Such a high price tag keeps these units out of the reach of many students, hobbyists, and researchers. However, for these people, the precision and speed provided by a professional oscilloscope may exceed their needs. Therefore, the goal of this project is to design a low-cost oscilloscope that could be used by groups or individuals operating on a low budget.

As indicated, the most important requirement for the oscilloscope is a low price. The target price for the oscilloscope is $100. Although $100 is decidedly not *cheap*, it should be well within the budgetary constraints of most research groups and hobbyists. The $100 budget includes the cost of all parts needed to assemble the oscilloscope that could not reasonably expected to be on hand.

Following price, the next most important requirement is a broad frequency support. Specifically, the oscilloscope should be able to display signals over at least the human auditory range. For the purposes of this project, that range will be defined as 20Hz–20kHz. Moreover, the oscilloscope must be capable of displaying a minimum of 10 samples per period over that frequency range. This range is sufficient to capture a number of common analog signals, including those from biological sensors, electromechanical sensors, and, of course, acoustic sensors.

The third important requirement is ease of assembly. Since the oscilloscope is targeted at students and hobbyists, it must be constructable without any expensive equipment or special skills. The only special tool necessary to assemble the board should be a soldering iron, and only a basic one at that. Although the oscilloscope does not aim to be a first-time soldering project, someone with only a little soldering experience or determination should be able to solder it.

Requirements Summary

1. Cost less that $100
2. Capture frequencies over the range 20Hz–20kHz
3. Constructable with a basic soldering iron and limited soldering skills

2. Possible Solutions

From a very high level, there are only two classes of possible solutions: design from scratch or reuse existing equipment. In the latter category, the most prominent choice is to use a standard PC with sound card. Using minimal external circuitry, the microphone or line-in jack could be used as a general purpose ADC (Analog-to-Digital Converter). With appropriate software, the signal could be captured and plotted on the computer monitor. Several commercial products exist that use this approach.

The PC solution has a number of drawbacks, however. The most significant is the frequency range. Since the sound card is specialized for capturing sound, it only samples at standard audio sampling rates (e.g. 44.1kHz). At 44.1kHz, it would not be able to meet the second requirement outlined previously. In particular, the sound card would only be able to capture signals up 4.41kHz at 10 samples per period. A second problem with the PC solution is possible damage. If the input signal exceeds the voltage tolerances of the sound card, the user risks damaging an expensive computer.

Besides the PC, no widely available consumer grade equipment exists with an easily accessible ADC. As such, the oscilloscope will need to be designed from the ground up. However, there are still a myriad of design options available. Clearly, any design must have at least three basic components: an ADC, a processing unit, and a display. To allow user interaction, some manner of input device is also highly desirable.

To meet the cost requirement, the processing unit must be inexpensive. A great many microcontrollers and embedded microprocessors fit this bill. Although satisfying the cost requirement, embedded microprocessors (such as ARMs) generally fail the ease of construction requirement. These devices are invariably available only in small surface-mount packages that are difficult if not impossible to solder by hand. This leaves microcontrollers as the only viable processing unit. Among microcontrollers, the two most recognizable lines are the AVRs from Atmel and the PICs from Microchip. Although either line could probably be used, the AVRs are supported by a wide variety of high-quality open source (and therefore *free*) software development tools. In contrast, quality tools for the PICs are commercial products.

Many AVR microcontrollers include an internal ADC that could, at first glance, provide the necessary ADC functionality. Unfortunately, further inspection reveals that the internal ADC is only capable of sampling at about 15kHz. This is decidedly worse than the PC solution described above. Instead, an external ADC must be used. From a high level design standpoint, a precise ADC choice is relatively unimportant with the caveat that it meet the design requirements. Specifically, it should have maximum throughput of at least 200kHz (to meet the second requirement), and be available in easily soldered package (to meet the third requirement).

The display options are even more varied than the CPU options. Three prominent choices are a dedicated graphic LCD unit, a computer monitor, or a television. A small graphic LCD would be the best choice from a usability standpoint. It would allow the oscilloscope to be small and portable, as well as allow for relatively simple microcontroller graphics software. Unfortunately, graphic LCDs big enough to be useful (3–4in) are outside the budgetary constraints ($80–150), and graphic LCDs cheap enough (less than $30) are too small (less than 2in). Computer monitors can be ruled out immediately because a microcontroller is simply not powerful enough to generate a VGA graphics signal.

A television is left as the most plausible display. The most significant advantage to using a television is that it can be considered “on hand” (since most of the target audience will have easy access to one) and thus excluded from the budget. Televisions also require no assembly, which is in line with the third requirement. Composite NTSC signals are entirely possible to generate on a microcontroller, and thus a television is viable from a software perspective. The biggest disadvantage of a television is its bulkiness. However, if the size of the television is a problem, portable televisions with 3–5in screens can be found for less than $50 (although this is not counted in the budget).

The primary input devices on commercial oscilloscope are knobs and buttons. These provide for a simple and intuitive interface. Optimally, this project would replicate that interface. Unfortunately, knobs are relatively expensive (several dollars apiece), bulky, and somewhat more complex to implement reliably. Buttons, however, are cheap (less than $0.25 apiece), small, robust, and simple to implement. Furthermore, the user interface requires buttons (e.g. run/stop), but not knobs (increment and decrement buttons can be used in lieu of knobs). Given the general constraints of the oscilloscope, the improved user interface offered by knobs does not adequately counteract the added complexity.

Proposed Solution

- Processing unit: Atmel AVR microcontroller
- ADC: external 200kHz or greater external ADC
- Display: NTSC compatible television
- User interface: buttons

3. Background

3.1. NTSC Black and White Television Signal

As discussed, the primary output device for the oscilloscope is any standard NTSC television set. NTSC, originally developped in the 1940's, is the analog signal used to carry television throughout North America (as well as a number of other nations). Although the current NTSC standard incorporates color, it is completely backwards compatible with the older black and white standard: any television set capable of displaying a color NTSC signal will also display a black and white NTSC signal.

In NTSC, every frame of video is 525 lines long (of which only 485 contain image data), and frames are updated at 29.97 frames per second. However, since NTSC is interlaced, each frame consists of two fields: *even* and *odd*, which encode the even and odd lines, respectively. The actual NTSC signal is therefore a sequence of alternating even and odd fields, with each field encoding 262.5 lines. A special pulse called a horizontal sync separates each line, and complex sequence of pulses called the vertical sync separates each field.

Despite the seeming complexity of the NTSC standard, it turns out that most television sets are quite forgiving. Not only is timing flexible (as long as it's *consistent*), but the format of the fields is rarely strictly enforced. Therefore, it is possible to generate a much simpler black and white signal.

The most important simplification is regarding interlacing: simply don't do it. Instead, produce identical frames of exactly 262 lines each. Furthermore, the vertical sync can be dramatically simplified. By inverting the horizontal sync during lines 248–250, a vertical sync interpretable by the television can be generated.

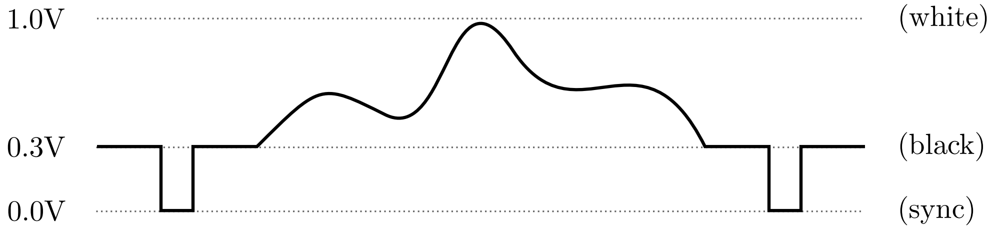


Figure 1: Single line of NTSC frame.

3.2. Generating NTSC Video Signal

The simplified NTSC video signal described in the previous section can be generated with an 8-bit microcontroller. Traditionally, this has been accomplished via "bit-banging", that is, manually turning on and off I/O pins on the microcontroller. However, using newer Atmel AVR microcontrollers, it is possible to generate the video part of the signal in hardware, leaving only the syncs to be generated by software.

Since the video signal being generated is strictly black and white, and encoded at one bit-per-pixel, the signal can be thought of as a stream of bits being "shifted out" of the microcontroller. It is clear, then, that a shift-register clocked at the pixel frequency would be sufficient for generating the video signal. Most microcontrollers have such a shift-register built in: the SPI bus. Data transmission on an SPI bus is accomplished by shifting bytes out at the SPI clock frequency.

Unfortunately, the hardware SPI controller in most microcontrollers is not buffered. This means that a byte cannot be loaded into the SPI module until the previous byte has been completely transmitted. Even with carefully designed software, there must necessarily be a delay between the end of one byte and the beginning of the next. This delay will show up on the television as a blank space every eight pixels. Since the pixel clock is usually within an order of magnitude of the CPU clock, a delay of even a few CPU cycles could result in a blank space a pixel wide!

This problem can be circumvented in the newest AVR microcontrollers. The USART (Universal Synchronous/Asynchronous Receiver/Transmitter) in these microcontrollers have a new SPI compatibility mode. By configuring the USART to use this new mode, the USART becomes a fully-compatible SPI master controller. Unlike the dedicated SPI controller, however, the USART is buffered. Since the next byte to be transmitted can be loaded before the current byte is complete, a continuous series of bits can be shifted out uninterrupted.

4. Hardware Design

4.1. Power

Input power is supplied to the oscilloscope via a standard 2.1mm barrel connector. The oscilloscope will accept 9-15V DC, tip-positive power. The positive terminal is connected to a single pole, dual throw (SPDT) switch. When the switch is in the "on" position, the input voltage is connected to a voltage regulator. The regulator is a D-PAK surface mount version of the ever-popular 7805 5V regulator. The D-PAK package is large enough to be easily soldered by hand. All other electronics are powered from the output of the regulator. A green LED with a 330Ω load resistor is also directly connected to the regulator. This provides clear feedback to the user about whether the oscilloscope is turned on.

4.2. Microcontrollers

Both microcontrollers used in the oscilloscope are ATmega644's from Atmel. This microcontroller was selected based on the constraints of the video microcontroller. In particular, they possess a USART with the SPI-compatible mode mentioned previously. As indicated, this is exceedingly useful for generating video because video pixels can be shifted out by hardware rather than bit-banged in software. Moreover, the ATmega644 sports 4KiB of RAM, enough to hold a 200 × 128 pixel raster image. A raster of that size would not fit in the 2KiB of RAM on the older ATmega32 (also considered).

The ATmega644 also satisfies the first and third (cost and ease of assembly) requirements for the oscilloscope. At about $8 apiece, the two microcontrollers fit well within the budget. Although superior to the older ATmega32 in every metric, the ATmega644 actually costs a few cents less than the older ATmega32. Although the pennies saved are not a major factor, they do eliminate any incentive to use the ATmega32. The ATmega644 is available in DIP package, which is easy to solder, even for beginners.

The microcontrollers are connected together via an SPI bus. The MOSI, MISO, and $\overline{\text{SS}}$ pins of the two microcontrollers are directly connected. The SCK pins are also connected when the PROG_JMP jumper (discussed more later) is set to "Normal". Pins 2 and 3 on port B of each microcontroller are also directly connected to the same pins on the other microcontroller. These lines are used for a simple hand-shaking procedure, and are named ACC and REQ, respectively.

Each microcontroller operates from an independent 20MHz crystal oscillator connected to XTAL1 and XTAL2. Those two pins are also connected to ground through 20pF capacitors. Both microcontrollers have a green LED connected to pin 7 on port D. The LED's have 330Ω load resistors to limit current. These LED's have no specific meaning, and were intended primarily as a debugging aid.

The video microcontroller has pin 4 on port D is tied to ground. This was intended as an easy means to differentiate the microcontrollers in software. The same pin on the analog microcontroller has a pull-up resistor (as part of the ADC interface). Therefore, that pin will be read as 0 on the video microcontroller and 1 on the analog microcontroller. Upon initialization, the software can check that pin and flash a warning on the LED if it is executing on the wrong microcontroller.

4.3. Video DAC

As previously discussed, to generate a black and white NTSC signal, three voltages are required: 0V, 0.3V, and 1V. These voltages can be encoded on two I/O pins on the microcontroller, namely VIDEO and SYNC. When both pins are low, a sync pulse is being generated, requiring 0V. When

SYNC is high, then the VIDEO pin determines the color: high should be white (1V) and low should be black (0.3V). The voltage levels are summarized in Table 1.

VIDEO	SYNC	Meaning	V_{out}
0	0	sync	0V
1	0	*invalid*	
0	1	black	0.3V
1	1	white	1V

Table 1: VIDEO and SYNC truth table

The voltage levels can be generated using a simple three-resistor DAC (Digital-to-Analog Converter). Resistors R1 and R2 are connected to the VIDEO and SYNC pins on the microcontroller, respectively, with their other ends tied together. Resistor R3 then connects resistors R1 and R2 to ground. The value of R3 is chosen to be 75Ω, which is the characteristic impedance of standard video coaxial cable. Equations 1 and 2 relate the resistor values to V_{white} and V_{black}, respectively.

$$V_{white} = \frac{R3}{R1||R2 + R3} V_{OH} \tag{1}$$

$$V_{black} = \frac{R1||R3}{R1||R3 + R2} V_{OH} \tag{2}$$

Assuming $V_{OH} = V_{DD} = 5$V, and given that R3 is 75Ω, the above equations yield $R1 = 430\Omega$ and $R2 = 1\text{k}\Omega$. However, as with the timing of the NTSC signal, there is some flexibility with the voltage levels. As such, a value of 330Ω was used for $R1$ to reduce the number resistor values used (330Ω is also used as load resistors for the LED's).

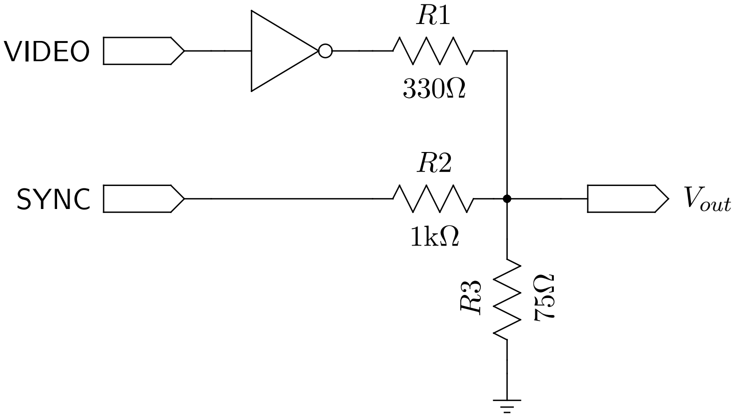


Figure 2: Schematic of video DAC.

The sync pin is connected to a normal I/O pin on the microcontroller. This pin is controlled directly by software: set high normally and low during a horizontal sync (and vice versa during vertical sync). The video pin, however, is connected via an inverter. This is necessary because it is connected to the TXD pin on the microcontroller. The TXD pin is the transmit pin for the microcontroller's internal USART. As mentioned (and discussed in more depth later), the USART is configured in SPI mode, which allows it to act as a high-speed shift register. Unfortunately, when configured as such, the TXD line is pulled high when idle. Since the video pin should be low by default, the inverter is needed. This has the side effect of also inverting the video color, which is compensated for in software.

The resulting analog signal coming from the video DAC is connected to a standard RCA style connector. Any 75Ω video cable can then be used to connect the oscilloscope to the composite input on an NTSC television.

4.4. Analog Frontend

The analog signal enters the oscilloscope via a standard 1/8in mono audio jack. The signal is conditioned in the so-called "analog frontend". This circuit is responsible for amplifying the signal as well as providing rudimentary protection. The incoming signal is clamped by a pair of Schottky diodes. One diode connects the signal to ground, and the other to V_{DD}.

After the diodes, the signal is passed into a programmable gain amplifier. The amplifier is Microchip MC6S21 PGA that is configurable via SPI. It has gain settings of 1, 2, 4, 5, 8, 10, 16, and 32 V/V. By using the PGA to amplify the signal in the analog domain, the precision of the digitization is constant across all V divisions in the oscilloscope.

After passing through the PGA, the signal is passed through a 1/2 attenuator. This is because the ADC chosen only has an input range of 2.5V, and the oscilloscope's accepted voltage input range is 5V. By having an explicit analog attenuator, the oscilloscope can treat the values coming out of the ADC as if they were on a 0–5V scale. The attenuator is simply a voltage divider followed by non-inverting buffer. The non-inverting buffer is build from a Microchip MCP601 rail-to-rail output single-supply op amp

The entire analog frontend is referenced to a 2.5V virtual ground. The virtual ground is generated by a Microchip MCP1525 precision 2.5V reference. The virtual ground means that incoming signals in the range of 0–5V referenced to real ground are treated as if they were signals in the range -2.5–2.5V referenced to virtual ground. Further more, the input signal is DC coupled, which means that signal must strictly fall within the 0–5V range (referenced to real ground), and should not have a DC offset.

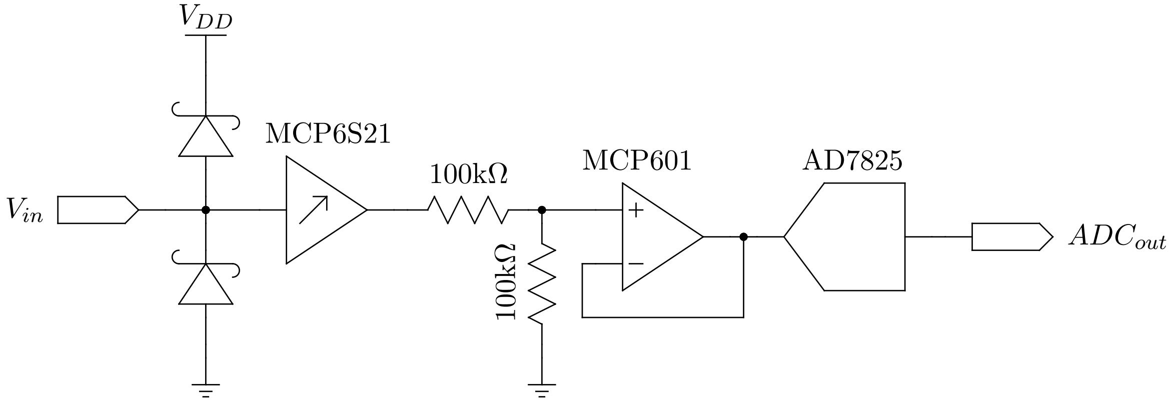


Figure 3: Schematic of analog front end.

4.5. ADC

After signal conditioning in the analog frontend stage, the signal is passed to the ADC. The ADC used is the Analog Devices AD7825. This ADC is a high-speed parallel ADC that can operate at 5V. Although able to operate at 5V, the ADC is limited to a 2.5V-wide input range. The range may appear anywhere within the 5V operating range. The precise input range accepted by the ADC is $V_{mid} - 1.25$V to $V_{mid} + 1.25$V. For the oscilloscope, V_{mid} was set to 2.5V, which is the virtual ground generated in the analog frontend. In order to minimize noise, all unused input channels to the ADC (V_{IN2}–V_{IN4}) are connected to ground.

The ADC has eight data lines and several control lines. The data lines DB8–DB0 are connected to port C of the analog microcontroller. The $\overline{\mathsf{PD}}$ control line is connected to V_{DD} to ensure that the ADC never enters power down mode, and the $\overline{\mathsf{CS}}$ line is connected to ground so that the ADC is always selected. Both channel select lines, A0 and A1, are connected to ground, permanently selecting channel 1. The remaining control lines, $\overline{\mathsf{CONVST}}$, $\overline{\mathsf{RD}}$, and $\overline{\mathsf{EOC}}$ are connected to port B on the analog microcontroller.

According to the ADC datasheet, it is important that the ADC power up with $\overline{\mathsf{CONVST}}$ pulled high to ensure proper operation. Since the pin on the microcontroller will remain in Hi-Z state until explicitly set as an output in software, $\overline{\mathsf{CONVST}}$ will be floating for several milliseconds after power is connected. Therefore, a 10kΩ pull-up resistor connects $\overline{\mathsf{CONVST}}$ to V_{DD}. This ensures that the pin is held high until the microcontroller finishes booting.

4.6. In-circuit Programming

During normal operation of the oscilloscope, the two microcontrollers are connected together via SPI, with the analog microcontroller configured as the master. Since in-circuit serial programming (ISP) also occurs over the SPI bus, this causes a conflict at the video microcontroller. When attempting to program the video microcontroller, the programmer would not be able to establish a connection since the analog microcontroller was controlling the bus.

By connecting the reset pins of the two microcontrollers together, while programming one the other will be held in reset mode. This causes all of its pins to go into a high-impedance state, which allows the programmer to take ownership of the SPI bus. Unfortunately, since both microcontrollers are connected to the SPI bus, their would be no way of programming the microcontrollers individually.

According to Atmel appnote AVR042 "AVR Hardware Design Considerations", multiple microcontrollers connected to the same programming interface can be programmed individually by selectively connecting the SPI clock. Although both microcontrollers will be held in reset, and both will receive the data, only the microcontroller receiving the SPI clock will be programmed. On the board, this functionality is provided by the PROG_JMP jumper block. By placing a jumper on "normal", the SPI clocks of the two microcontrollers are connected to each other. On "video", the SPI clock of the video microcontroller is connected to the ISP clock. On "analog", it's the analog microcontroller's SPI clock that is connected to the ISP clock.

4.7. Printed Circuit Board

A custom PCB (Printed Circuit Board) was designed to hold all the electronics. The board size chosen was 3.8in × 2.5in. This size was chosen because it is the size of the ExpressPCB MiniBoard. The MiniBoard is two-layer board without soldermask or silkscreen. A standard order of three MiniBoards costs only $51.

Unfortunately, ExpressPCB will not accept MiniBoard orders designed in any software but their own, which lacks important features (like ratlines) and is otherwise difficult to use. The boards were therefore designed using the open source tools gEDA and (the aptly, yet unfortunately, named) PCB to design the schematics and PCB, respectively. Although this rules out using the ExpressPCB service, the size constraints were maintained in order to allow the possibility of converting the oscilloscope to a MiniBoard in the future.

The PCB is arranged for maximum usability. The signal input, video output, and power jacks are all placed along the back of the board. The power switch is located right next to the power connector, and the power LED is intuitively located right next to that. The user interface buttons are spaced evenly across the front of the board for easy access.

The analog frontend is located along the left side of the board. The analog components are grouped together to minimize the distance the analog signal needed to travel. Separate analog power and ground rails are used to power the analog circuit. These rails are connected to the main power and ground rails at only a single point and filtered with a capacitor. The sperate power and ground rails are intendend to reduce the noise generated by the digital components that reaches the analog components. Furthermore, an analog ground plane runs under the entire analog frontend to further minimize noise on the analog signals.

The microcontrollers are adjacent to each other and oriented in the same direction. This allows their SPI pins to be connected with short, straight traces. The microcontrollers are intendended to be put into 40-pin DIP sockets. Therefore, the space under the microcontrollers is utilized for various surface mount resistors and capacitors. The debugging LEDs are placed near their corresponding microcontrollers so that it is obvious which LED corresponds to which microcontroller.

The ISP header and PROG_JMP jumper are placed along the back of the board so that programming cables would not interfere with the user interface during software development. The ISP header was oriented such that when the oscilloscope was placed adjacent to an STK-500, it would be oriented the same as the ISP header on the STK-500. The PROG_JMP jumper was oriented so that the position of the jumper correlated spatially with the microcontroller being programmed. Placing the jumper on the position closest to the back of the board allows programming of the microcontroller closest to the back of the board (the video microcontroller).

5. Software

5.1. Analog Microcontroller

5.1.1. User Interface

In addition to the television, which is handled by the video microcontroller, the user interface consists of eight buttons. The user can manipulate the oscilloscope's four operating parameters (trigger, time division, voltage division, and cursor) using these buttons. The buttons available to the user are summarized in Table 2.

Button	Function
RUN	Stops/starts the oscilloscope
SEQ	Arms a trigger when stopped
TRIG	Sets current parameter to trigger
TIME	Sets current parameter to time division
VOLT	Sets current parameter to voltage division
CRS	Sets current parameter cursor
DOWN	Increases value of current parameter
UP	Decreases value of current parameter

Table 2: Button descriptions.

The RUN button simply stops the oscilloscope if it is running and starts it if not. When stopped, the last displayed waveform is left on the screen. Although the trigger, time division, and voltage division parameters can be adjusted while the oscilloscope is stopped, they do not take effect until the oscilloscope is started, or a single sequence is armed. In other words, it is impossible to zoom in on a frozen waveform. The cursor is only available when the oscilloscope is stopped.

The SEQ button arms a trigger when the oscilloscope is stopped. After the trigger is armed, the oscilloscope will wait for a trigger condition. When a trigger condition occurs, the display will be updated with a new waveform. The oscilloscope will remain stopped until SEQ is pressed again, or until it is started again by pressing RUN.

The TRIG, TIME, VOLT, and CRS buttons change the active parameter to trigger, time division, voltage division, and cursor. The active parameter can then be adjusted using the UP and DOWN buttons.

Because the buttons bounce when pressed, a software debouncer is needed to prevent a single button press from being registered multiple times. The debouncer works by periodically polling the buttons and storing them in a circular buffer. A button is considered "pressed" if and only if it is pressed in each element of the circular buffer.

The debouncer is run off the timer 2 OC2A interrupt. Timer 2 is configured in CTC (Clear Timer on Compare math) mode. In that mode, it the counter resets when it reaches the value in register OCR2A. Additionally, the OC2A interrupt is enabled, which fires whenever the counter reaches OCR2A. The clock timer is set to $f_{osc}/1024$ ($\sim$19.5kHz), and the OCR2A register is set to 195. This results in a time base of $\sim$10ms.

The OC2A ISR reads the current value of PINA (the buttons), and stores it to the current index of the circular buffer. The buffer index is then incremented. The length of the circular buffer is a power of 2. This means that the index never needs to be reset to zero manually. Instead, it is merely masked. For example, if the length of the circular buffer is 4, then the buffer index is masked by 0x03.

When the main loop needs the button values, it merely bit-wise ands all the elements of the circular buffer. A button is pressed if its corresponding bit in the result is 1. Furthermore, the previous such result is stored. A button has *just* been pressed if its corresponding bit in the result is 1 *and* its corresponding bit in the previous result was 0. The main loop checks to see if each button in turn has *just* been pressed, and acts on it accordingly.

5.1.2. Managing Sample Rate

To minimize the workload on the microcontroller, ADC samples are acquired only as needed. This means that the sample rate varies depending on the current time division setting. Specifically, the sample rate is $T/200$ samples/s, where T is the current time division setting. The sampling rates are divided into three "speed grades". In the slow speed grade, samples are drawn to the screen as soon as they are read. This is used for the long time divisions (0.5–5s). In the medium and fast speed grades, the screen is only updated twice per second maximum. The medium speed grade is used for the 10ms–0.2s time divisions, and the fast speed grade is used for the 0.5–5ms time divisions.

The sample rate is controlled by timer 1, which is configured in CTC (Clear Timer on Compare match) mode. In that mode, the counter resets when it reaches the value in register OCR1A. Additionally, the OC1A interrupt is enabled, which fires whenever the counter reaches OCR1A. The timer clock is set to either $f_{osc}/1$ or $f_{osc}/8$ depending on the time division setting, since neither clock divider has sufficient range to cover all time divisions. Therefore, by adjusting OCR1A and the timer 1 clock appropriately, arbitrary time-bases can be defined. Table 3 summarizes the clock divider and OCR1A settings for all available time divisions.

Speed	Time	Clock	
Grade	Division	Divider	OCR1A
slow	5s	8	62500
	2s	8	25000
	1s	8	12500
	0.5s	8	6250
medium	0.2s	1	20000
	0.1s	1	10000
	50ms	1	5000
	20ms	1	2000
	10ms	1	1000
fast	5ms	1	500
	2ms	1	200
	1ms	1	100
	0.5ms	1	50

Table 3: Clock divider and OCR1A values for each time division.

5.1.3. ADC Conversion

The ADC has three important control lines ($\overline{\text{CONVST}}$, $\overline{\text{EOC}}$, and $\overline{\text{RD}}$) connected to port B (pins 4, 3, and 5, respectively), and eight data lines connected to port C. The microcontroller initiates a conversion by pulsing $\overline{\text{CONVST}}$ low and waiting until $\overline{\text{EOC}}$ pulses low, indicating that the conversion is complete. To read the data the microcontroller sets $\overline{\text{RD}}$ low, reads the data on PINC, and sets $\overline{\text{RD}}$ high again.

The minimum pulse width for $\overline{\text{CONVST}}$ is 20ns, as specified in the ADC datasheet. At 20MHz, the clock period for the microcontroller is 50ns. Therefore, the $\overline{\text{CONVST}}$ line needs to be pulled low for only a single CPU clock cycle. This is trivially accomplished by setting the pin low and immediately setting it high again.

Upon completion of a conversion, the ADC pulses $\overline{\text{EOC}}$ low for only 70–110ns, equivalent to only one or two CPU clock cycles. If the software were to simply poll the pin and wait for it to be low, it would likely miss the pulse entirely. To solve this problem, the $\overline{\text{EOC}}$ pin is connected to external interrupt 1, which is configured to trigger on a falling edge. External interrupts are detected asynchronously on the ATmega644, so an $\overline{\text{EOC}}$ pulse will be detected regardless of the software state. The microcontroller datasheet specifies that the minimum pulse width to be detected by an external interrupt is 50ns, which is less than the minimum pulse specified by the ADC.

Unless $\overline{\text{RD}}$ is pulled low, the ADC does not drive the data lines in order to allow multiple devices to share the same data bus. Unlike $\overline{\text{CONVST}}$ and $\overline{\text{EOC}}$, $\overline{\text{RD}}$ is not pulsed; it must be held low for the duration of the data read. The data set-up time for the ADC is 10ns, meaning $\overline{\text{RD}}$ needs to be low for that duration before attempting to read. Given that this is considerably less than the CPU clock period, the data can be read the cycle immediately after $\overline{\text{RD}}$ is set low. $\overline{\text{RD}}$ need only be held low for a minimum of 30ns, so it can be set high as soon as the data is read.

As previously discussed, the timer 1 OC1A interrupt is used to set the sample rate. In all speed grades, conversions are initiated directly by that ISR. Since the OC1A ISR only needs to pulse $\overline{\text{CONVST}}$ as described above, it is written in assembly for efficiency. In fact, the entire interrupt routine is only three instructions long:

```
cbi 0x0B, 4
sbi 0x0B, 4
reti
```

The first instruction sets pin 4 of port B low, and the second instruction sets the pin high again. 0x0B is the I/O address of the PORTB register. The third instruction returns from the interrupt. The ISR is defined with the ISR_NAKED flag, which instructs the compiler not to generate prologue or epilogue code. This is safe because no general purpose registers are written, and neither the `cbi` nor `sbi` instructions modify SREG.

The manner in which the data from the ADC is read differs depending on the speed grade. In the slow and medium speed grades, the sample is read in the external interrupt 1 ISR. When the ADC pulses $\overline{\text{EOC}}$, the interrupt fires. The ISR saves the current sample to a global variable, and sets a global software flag indicating a sample has been acquired. Upon seeing the sample flag set, the main loop clears the flag and processes the new sample. In the slow speed grade the sample is immediately and unconditionally sent to the screen.

In the medium speed grade, reading happens in cycles. During a read cycle, the new sample is checked against the trigger condition. When a trigger condition occurs, samples are stored to a buffer until 200 samples have been acquired. After the 200th sample, the samples are sent to the screen. Samples are then ignored for half a second before a new read cycle is initiated.

In the fast speed grade, the aforementioned algorithm is too slow. Instead, the external interrupt 1 hardware flag is polled manually. *All* interrupts except the timer 1 OC1A interrupt are disabled. When the ADC pulses $\overline{\text{EOC}}$, the external interrupt 1 flag is set, but the interrupt does not fire. The interrupt flag is detected in the main loop, and the sample is read and processed there. From that point, samples are processed in essentially the same manner as with the medium speed grade:

after a trigger condition, 200 samples are stored to a buffer and sent to the screen only after all 200 samples have been acquired.

Since the timer 2 OC2A interrupt is disable during a read cycle, the button debouncer is necessarily halted. In the fast speed grade, this is acceptable since an entire read cycle will take less than 10ms (the rate of the button debouncer). However, if a trigger condition never occurs on the input signal, then the microcontroller will become unresponsive while waiting for one. To prevent this and ensure a finite-time read cycle, after 200 samples without a trigger condition, the read cycle is terminated and restarted a half second later.

Although the medium and fast speed grades are functionally very similar, they are implemented differently for human-interface reasons. The medium speed grade is interrupt-driven, so the read cycle does not interfere with the button debouncing state machine. Because the read cycle in the medium speed grade is 10ms and slower, halting the button debouncer (as in the fast speed grade algorithm) would interfere with the user interface. For this reason, the fast speed grade algorithm is not applied to the medium speed grade.

5.1.4. Setting Gain

The voltage division of the oscilloscope is handled entirely by changing the gain of the PGA. The PGA is connected to and controlled via the SPI bus interface on the microcontroller. The PGA has two parameters that can be controlled: channel and gain. Since the PGA used on the oscilloscope only has a single channel, the first parameter can be safely ignored.

The PGA is controlled by sending it two bytes over the SPI bus. The first byte is the instruction, and the second byte is either the gain or channel. Bits 7:5 in the instruction byte specify the command. The PGA recognizes only two commands: enter shutdown mode (001) and write to register (010). Bit 0 of the instruction byte is the address bit. When the command is write to register, the address bit specifies whether to write to the gain register (0) or the channel register (1). Bits 2:0 in the second byte are the gain or channel selection, depending on the address bit. The only command the microcontroller ever needs to send to the PGA is write to gain register. Therefore, the first byte sent is always 0x40.

The algorithm for sending a gain selection to the PGA is quite simple. First, the PGA's $\overline{\text{CS}}$ line is set low. The instruction byte is then loaded into the SPI data register. The software stalls until the SPI controller indicates that transmission is complete. The gain selection is then loaded into the SPI data register, and again the software stalls until transmission is complete. Finally, $\overline{\text{CS}}$ is set high again.

Since sending a command to the PGA blocks execution, it is advantageous to set the SPI clock as fast as possible. The PGA datasheet specifies the maximum clock frequency as 10MHz. The maximum SPI clock frequency that the microcontroller can generate is $f_{osc}/2$, which is also 10MHz. However, since the SPI bus is also shared with the other microcontroller, the SPI clock frequency should remain within it's limits as well. The ATmega644 datasheet specifies that when configured as slave, the SPI is only guaranteed to work with an SPI clock up to $f_{osc}/4$ (5MHz). Therefore, the SPI clock is configured to be $f_{osc}/4$.

The PGA can operate in SPI modes 00 and 11. Since both microcontrollers can be configured to any of the four SPI modes, the PGA represents the limiting factor. Therefore, SPI mode 00 was chosen arbitrarily between the two modes the PGA supports. In SPI mode 00, the first (leading) edge of each byte is a positive edge. Received data is sampled on the rising edge and transmitted data is set up on the falling edge. The SPI controller was also set to master mode.

The PGA requires a minimum of 40ns between the falling edge of $\overline{\text{CS}}$ and the first SPI clock edge. Since this is less than the CPU cycle time of 50ns, data can be loaded into the SPI data register in the instruction immediately after $\overline{\mathsf{CS}}$ is set low. The PGA also requires a minimum of 30ns between the last SPI clock edge and the rising edge of $\overline{\mathsf{CS}}$. Again, since this is less than the CPU clock period, $\overline{\mathsf{CS}}$ can be set high immediately after the SPI transmission is complete.

5.1.5. Data Transmission to Video Microcontroller

The analog microcontroller is connected to the video microcontroller via the SPI bus. As previously discussed, the SPI controller was configured as master, SPI mode 00, with a clock of $f_{osc}/4$. Data transmission to the video microcontroller is very similar to data transmission to the PGA: the video microcontroller's $\overline{\mathsf{SS}}$ pin is pulled low, the data is transmitted, and the $\overline{\mathsf{SS}}$ pin is pulled high again.

However, sending data to the video microcontroller has additional overhead in the form of request (REQ) and accept (ACC) lines from the video microcontroller. Because video generation is very time critical, the video microcontroller may not always be able to service the SPI data. The REQ and ACC lines provide a form of handshaking to allow the two microcontrollers to coordinate.

When the analog microcontroller is ready to send data to the video microcontroller, it must first wait for the ACC line to be high. It can then request permission to send data by setting REQ low. When the video microcontroller is ready, it will pull ACC low. At this point, the analog microcontroller is free to send a single byte to the video microcontroller. When transmission of that byte is complete, the analog microcontroller will set the REQ line high and continue execution. Some time after REQ is set high, the video microcontroller will set ACC high.

As noted, unlike communication with the PGA, data is transmitted to the video microcontroller one byte at a time. This works out well because most of the data sent to the video microcontroller are samples, which can be packaged in a single byte. Bytes sent to the video microcontroller are divided into two categories: commands and data. Commands and data are typically distinguished by bit 7: 1 for commands and 0 for data.

For commands, the high nibble is the op-code, and the low nibble is optional data. The defined commands are summarized in Table 4. The meaning of data bytes is dependent on the current data-reception mode in the video microcontroller. The data reception mode can be set using the MODE command. Mode 0 is normal (samples), 1 is header text, 2 is header text, 3 is cursor, and 4 is trigger.

Op-code	Name	Function
0x80	ZERO	Resets x-counter to 0
0x90	CLR	Clears screen and resets x-counter to 0
0xAm	MODE	Sets data-reception mode to mode m

Table 4: Video command op-code descriptions.

When in mode 0, the video microcontroller interprets data bytes as samples. It maintains its own x-counter, which it increments with each received data byte. The x-counter will not automatically reset to 0, and the video microcontroller will ignore further data when the x-counter reaches 200. To force the x-counter back to 0, the analog microcontroller must send either the ZERO or CLR commands. ZERO sets the x-counter to 0 without affecting the current waveform. CLR removes the current waveform before setting the x-counter to 0.

When in modes 1 or 2, the video microcontroller interprets the data bytes as characters. In mode 1, the characters are stored to the header string, and in mode 2 the characters are stored to the footer string. In either case, the video microcontroller expects *exactly* 25 characters. The analog microcontroller can set the contents of the header or footer by setting the mode to 1 or 2 (respectively), transmitting 25 characters, and setting the mode back to 0.

When in modes 3 or 4, the video microcontroller waits for a single data byte before returning to mode 0 automatically. In mode 3, the data byte is interpreted as a cursor position, and in mode 4, the data byte is interpreted as a trigger level. To set the cursor position, the analog microcontroller sets the mode to 3 and sends the cursor position. To set the trigger level, the analog microcontroller sets the mode to 4 and sends the trigger level.

5.2. Video Microcontroller

5.2.1. Character Look-up Table

The target resolution for the oscilloscope video was 200 × 150 pixels. This would have required 3750 bytes of SRAM to hold the raster, leaving a mere 346 bytes for all other data, including the stack. Since the top and bottom bars of the screen would only display text anyway, it made sense to save SRAM by not rendering those lines in the raster. If the characters could be generated on the fly while the scanline was being output, then the top and bottom bars could be stored as simple character arrays.

In order to successfully generate the characters at scantime, the character symbols must be stored in an efficient look-up table. Since the screen pixels are pushed out through the USART, it makes sense that the characters should be 8 bits wide. This means that each byte pushed to the screen represents a line of a single character. Eight pixel wide characters mean that each byte of a look-up table represents a different character. If the look-up table is sorted in ASCII order, then the ASCII code for each character is conveniently also that character's index into the look-up table.

Each line of the characters is stored in a separate look-up table. For example, the first lines of the characters are grouped together in one look-up table, and so on. By aligning each look-up table to a 256 byte boundary, and sorting the look-up tables by the line they represent, the address of a precise line of a specific character can be computed using simple concatenation. The high byte of the address is the address of the first character of the first look-up table, and the low byte is simply the ASCII character code. By adding the line number to the high byte, specific lines can be addressed.

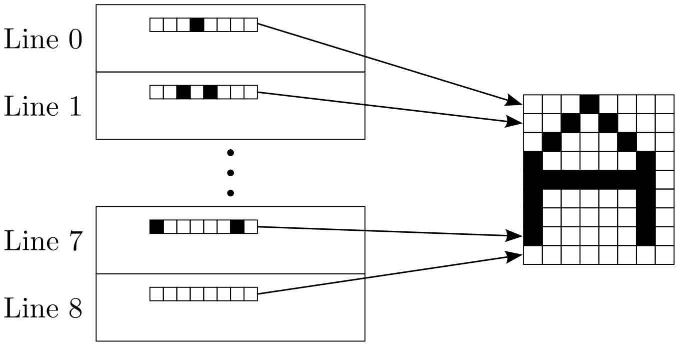


Figure 4: Organization of character look-up table.

5.2.2. Graph Raster

The main part of the oscilloscope screen is the graphing area for the waveform. This is stored as one bit-per-pixel raster. The graphing area is 200 × 128 pixels, which requires $(200 \cdot 128)/8 = 3200$

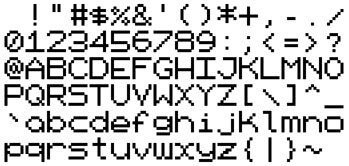


Figure 5: Entire set of displayable characters.

bytes of RAM. The raster is stored as 1-dimensional array of 3200 elements. Element 0 represents pixel (0,0) (top left) on the screen, and an increasing index corresponds to pixels across, then down. Because pixels are packed horizontally, each line of the display requires 25 bytes to store. This part of the code is derived from Bruce Land's TV code, altered to support the higher resolution and ported from the CodeVision compiler to the AVR-GCC compiler.

Because eight pixels are packed in a byte, indexing individual pixels is quite tricky. Setting or clearing a point (x,y) in the raster requires several steps. First, two indices need to be computed: i, the index of the byte in the raster array corresponding to pixel (x,y), and j, the bit-index into that byte corresponding to the pixel. Equations 3 and 4 show the mathematical relationship between i and j (respectively), and x and y

$$i = y \cdot 25 + \left\lfloor \frac{x}{8} \right\rfloor \tag{3}$$

$$j = x - 8 \cdot \left\lfloor \frac{x}{8} \right\rfloor = x \bmod 8 \tag{4}$$

Since the divisor in both equations is a power of 2, the division can be replaced by a shift left, and the modulo can be replaced by a bit-wise and. Thus, programatically, `i = y*25 + x>>3` and `j = x & 7`. Once i and j have been computed, byte i in the raster array needs to be loaded and a bit mask needs to be generated.

The mask is a byte of all zeros except the bit corresponding to the pixel. This could be accomplished by shifting a 1 left by j. However, the AVR architecture lacks a barrel shifter, so it can only shift left by one. Thus, shift left by j would require a control loop to shift left by one j times. Instead, a simple look-up-table is used. The look-up-table is 8 bytes long, where byte j is `1<<j`. If the pixel is being set, the byte from the raster array is bit-wise or'ed with the mask. If being inverted, the byte is xor'ed with the mask. If the pixel is being cleared, the byte is and'ed with the 1's-complement of the mask. The resulting byte is then stored back into index i of the raster array.

5.2.3. Scanline Generation

NTSC scanlines are generated from the timer 1 OC1A interrupt. Timer 1 is configured in CTC mode with a clock of $f_{osc}/1$ (20MHz). Both OC1A and OC1B interrupts are enabled. The OCR1A register is set to 1271. This means timer 1 will reset and the OC1A interrupt will be triggered once every 1271/20MHz = 63.55μs, which corresponds exactly to the specified line time for NTSC signals.

The OC1A interrupt first generates a horizontal sync by setting the SYNC pin (port D pin 2) low for 5μs, and then high again. If the current line is within the visible portion of the screen, the corresponding line of image data is sent through the VIDEO pin (port D pin 1, TXD). Finally, the video line number is incremented and the state data for the next line is computed.

The image data is not sent manually, but rather through the USART. The USART is configured in MSPIM mode, in which it is a fully compliant SPI master controller. In this mode, the USART basically acts as buffered shift register. The USART clock is set to $f_{osc}/4$ (5MHz), which is also the pixel clock. The 25 bytes making up a line are loaded sequentially into the UDR0 register. After loading each byte, the software waits for the UDRE0 flag in the UCSR0A to be set, indicating that the USART transmit buffer is empty and ready for more data. As discussed earlier, a hardware inverter is necessary on the TXD line. To compensate, all bytes loaded into the UDR0 register must first be inverted in software. For simplicity, this inversion is implicit in the rest of this discussion.

Although the exact timing of the NTSC signal is flexible, it is imperative that the timing be *consistent*. The time between a pair of consecutive horizontal sync pulses must be exactly the same as between any other pair. Unfortunately, interrupts in the AVR architecture can only be serviced after the current instruction finishes executing. Since the number of cycles to execute an instruction varies depending on the instruction, the OC1A interrupt may be delayed by several cycles relative to previous or subsequent lines. At a 5MHz pixel clock, a single instruction corresponds to one quarter of a pixel! Since the delay of each line may be different than the other lines, and different than the same line in the previous frame, the image on the screen will appear to jitter.

To circumvent this problem the OC1A interrupt must always be entered in a consistent number of cycles. The only way to achieve this is to enter the interrupt from a sleep mode. The number of cycles the CPU takes to wake up and then service the interrupt is consistent, eliminating the jitter. The OC1B interrupt of timer 1 takes care of this. Register OCR1B is set to 1250, which causes the OC1B interrupt to fire about 1μs before OC1A. The OC1B ISR merely sets the global interrupt enable (which is cleared by hardware upon entering the interrupt) and then puts the CPU to sleep. Since the sleep is used strictly for timing, the CPU is configured to enter the lightest sleep mode (“idle”). When the OC1A interrupt fires, the CPU wakes up from sleep and executes the OC1A ISR.

The screen is divided into seven sections: header, header space, header rule, graph, footer rule, footer space, and footer. The header and footer sections are the text lines at the top and bottom of the image. The header and footer spaces are blank lines following the header and preceding the footer. The header and footer rules are white lines preceding and following the graph. The current section is part of the state data computed at the end of the preceding line.

The actual image data sent to the TV depends on the current section. The space and rule sections are the simplest. In the space section, no data at all is sent, resulting in a blank line. In the rule section, 25 bytes of 0xFF are transmitted, resulting in a white line.

The header and footer sections are the same, except that they refer to different character arrays. To generate an image line in the header or footer section, the software must know the current character line n, which is the current scanline minus the start of the header or footer section. This is used to index the n’th character look-up table as described above. 25 bytes of data are transmitted corresponding to the 25 characters in the header or footer character array. The ASCII code of each character is used as the index into the n’th look-up table. The byte from the look-up table is the byte transmitted.

In the graph section, the image data transmitted to the TV comes from the raster array. The software must know the raster line y, which is the current scanline minus the start of the graph section. 25 bytes of data from the raster array are transmitted, starting from index $y \cdot 25$. If the trigger level is visible, and y is equal to the trigger level, then 25 bytes of 0xFF are transmitted instead of the raster data.

5.2.4. Data Reception from Analog Microcontroller

Because the NTSC video signal is generated entirely in the OC1A interrupt, the main loop need only be responsible for drawing to the graph raster or setting the header and footer. The image on the screen is controlled entirely, though indirectly, by the analog microcontroller. Therefore, the image can only change when the video microcontroller receives data from the analog microcontroller.

Each iteration of the main loop starts by waiting for a request from the analog micrcontroller. When the REQ line is pulled low, the video microcontroller responds by setting the ACC line low. It then waits for an SPI transfer to be complete. At that time, it saves the data received from the analog microcontroller, and waits for REQ to be pulled high again. The video microcontroller then sets ACC high again before proceeding to process the data.

As previously discussed, the video microcontroller has five data reception modes: sample (0), header (1), footer (2), cursor (3), and trigger (4). If the reception mode is 0, 1, or 2, and bit 7 of the received byte is a 1, then the byte is processed as a command. If the reception mode is 3 or 4, or bit 7 is 0, then the byte is processed as data.

In mode 0, data bytes are samples to be displayed on the screen. The video microcontroller maintains an x-counter x for incoming sample data, as well as a buffer b of the current sample for each x-coordinate. If the x is less than 199, then the line from $(x, b[x])$ to $(x+1, b[x+1])$ is erased. The new sample then is stored to $b[x]$, and if x is greater than 0, the line from $(x-1, b[x-1])$ to $(x, b[x])$ is drawn. If the cursor is visible, it is redrawn in case it was damaged during the erasure of the old line. Finally, x is incremented. If x reaches 200, all further data bytes are ignored until a CLR or ZERO command resets x to 0.

In modes 1 and 2, data bytes are characters for the header or footer, respectively. Similar to mode 0, the video microcontroller maintains an index i for incoming characters. The index is initialized to zero upon entering mode 1 or 2. If i is less than 25, then the incoming character is stored to index i of the header or footer and i is incremented. Because the character image data is generated on the fly by the OC1A interrupt, simply storing the character to the header or footer is sufficient to be displayed. If i reaches 25, all further data bytes are ignored until the mode is changed.

In mode 3, the data byte is the cursor position. After receiving the cursor position, the mode is automatically reset to 0. This is because the cursor position can be greater than 128, which results in a 1 in bit 7. Therefore, when receiving a cursor position, it would be impossible to differentiate between a data and command byte. The cursor value 255 is a special case which indicates that the cursor should be turned off. If the cursor was on, the old cursor is erased from the screen. The sample segments connecting to the sample under the old cursor are redrawn. Finally, if the new cursor position is visible, it is drawn to the screen.

In mode 4, the data byte is the trigger value. Like mode 3, after receiving a trigger value, the mode is automatically reset to 0. Again, the special value 255 is used to turn the trigger display off. Because the trigger is drawn on the fly in the OC1A interrupt, merely setting the trigger value is sufficient for it to be displayed. The trigger level is not drawn to the graph raster, so no segments need to be redrawn.

6. Results

As discussed in detail in the Requirements section, the oscilloscope had three requirements: 1) cost less that $100, 2) capture frequencies over the range 20Hz–20kHz, and 3) constructable with a basic soldering iron and limited soldering skills. All three requirements were satisfactorily reached.

The total cost for all purchased parts for the oscilloscope is only $65.51, well within the target budget. Because shipping costs vary widely depending on location and shipping options, they are not included in the cost. However, standard ground shipping should be less than the remaining $34.49 of budget. As previously discussed, the television is considered "on hand" and not counted toward the budget. The ADC, PGA, op amp, and voltage reference are also not counted toward the budget. Free samples of these parts can be acquired directly from the manufacturer (Analog Devices for the ADC and Microchip for the others). Table 5 summarizes the costs of the various parts of the oscilloscope.

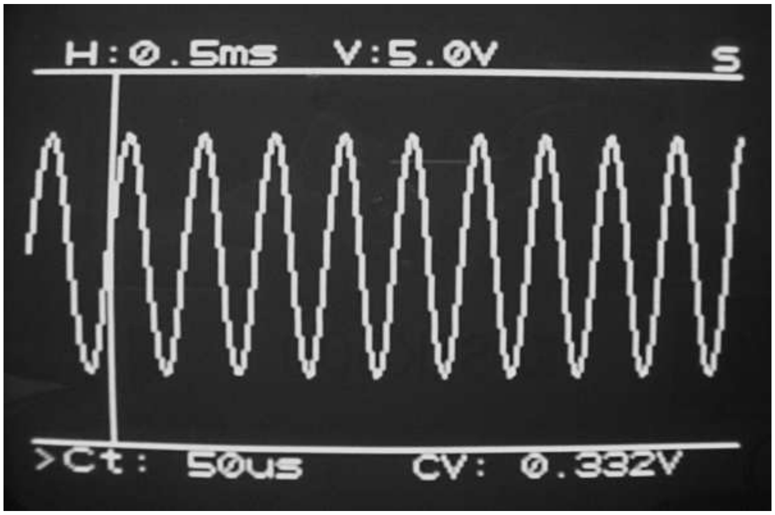
Parts	Cost
Printed circuit board	$33.00
Discrete components and IC's	$9.70
Connectors, buttons, and power switch	$7.07
Microcontrollers	$15.74
ADC, PGA, op amp, and voltage reference	–
Televsion	–
Total	**$65.51**

Table 5: Summary of oscilloscope cost.

All parts on the oscilloscope were chosen such that they could be soldered by hand. Where space permitted, through-hole packages were chosen over surface mount packages since the former are generally much easier to solder. The microcontrollers, PGA, op-amp, voltage reference, buttons, power switch, and all connectors were through-hole. For surface mount parts, the pins needed to be big enough and far enough apart that they could be soldered by hand. The resistors, capacitors, and LEDs are all in 1206 packages, which can be soldered with only minimal practice. The voltage regulator is in a D-PAK package, which has two large, widely spaced pins and wide solder tab. The ADC is in a SOIC package. The smallest components are the Schottky diodes and the logic inverter in SOT-23 and SOT-23-5 packages, respectively. These packages can be frustrating, but with a steady hand and a little patience can be soldered by hand.

Unfortunately, the boundaries of the ease of assembly requirement are stretched slightly by the microcontroller programmer. Although the physical oscilloscope can be assembled with just a soldering iron and a pair of tweezers, programming the microcontrollers requires a special piece of hardware, namely a programmer. However, given the target audience, it is very likely that if the user does not already own an Atmel programmer, he or she can likely find one to borrow. If not, simple programmers can be built or purchased for less than $5. Once the hardware has been acquired, the firmware can be downloaded to the microcontrollers using one of several free programs available for that purpose.

In the fastest time division (0.5ms), the oscilloscope samples data at 400kHz. Therefore, signals up to 40kHz could be displayed at 10 samples per period. 20Hz signals can be seen with at least 10 samples per period at the 1s time division and faster. This means that the frequency requirement is met completely: frequencies over the audio range (20Hz–20kHz) can be captured and displayed with at least 10 samples per period.


H:0.5ms V:5.0V S

>Ct: 50us CV: 0.332V

Figure 6: The oscilloscope screen with a captured 20kHz waveform.

Although the oscilloscope met or exceeded all stated requirements, it still has room for possible improvements. The most significant improvement is to add AC coupling. Since the analog frontend is referenced to a 2.5V virtual ground anyway, adding this feature would be quite trivial: three discrete components maximum. Specifically, a capacitor would need to be inserted in series between the input jack and the PGA. Immediately after the capacitor, high-impedance capacitors would need to connect the signal to ground and V_{DD} in order to center the signal at 2.5V. It is conceivable that the resistors may not even be needed.

Two other features that would be useful would be an external trigger and a serial terminal to dump data to a PC. Both of these features were originally planned. Although not discussed, the hardware for these features is already incorporated into the oscilloscope PCB. Unfortunately, time constraints prevented the features from being implemented in the microcontroller software. These features could be implemented with just a software update.

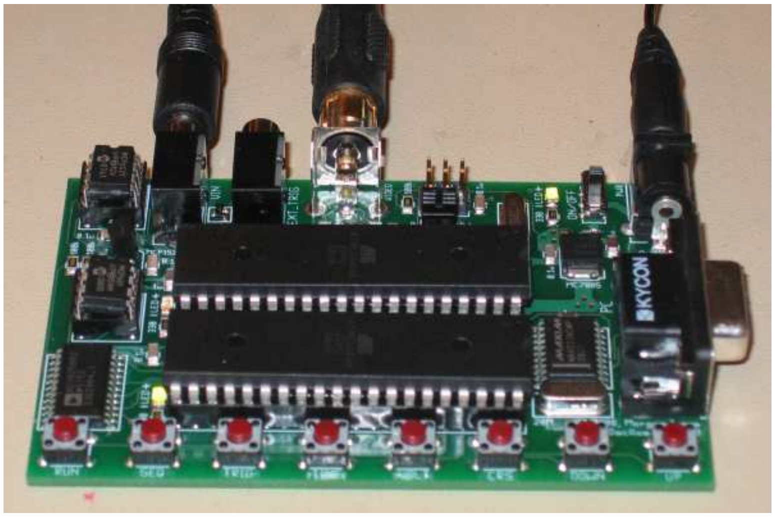
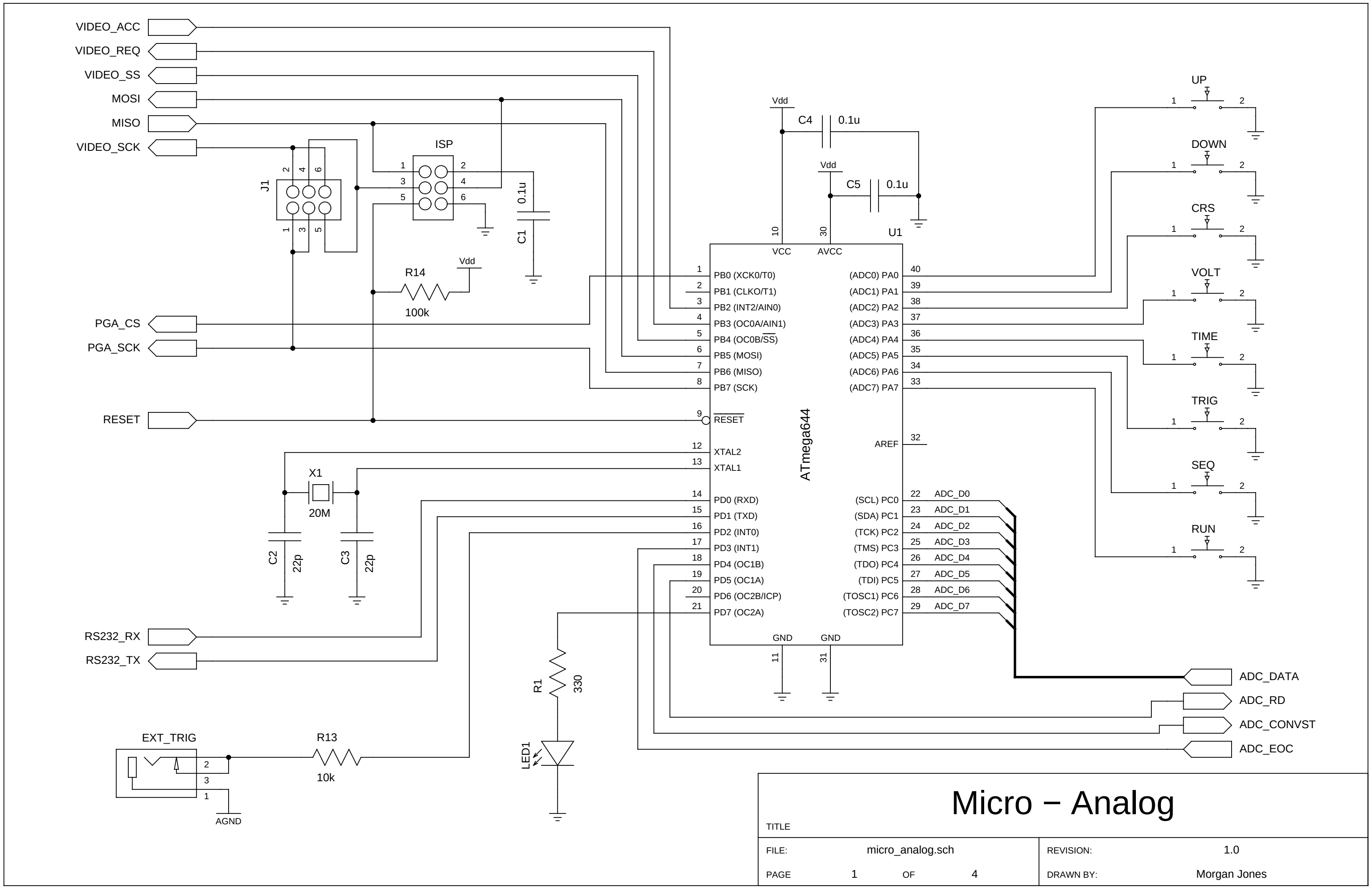
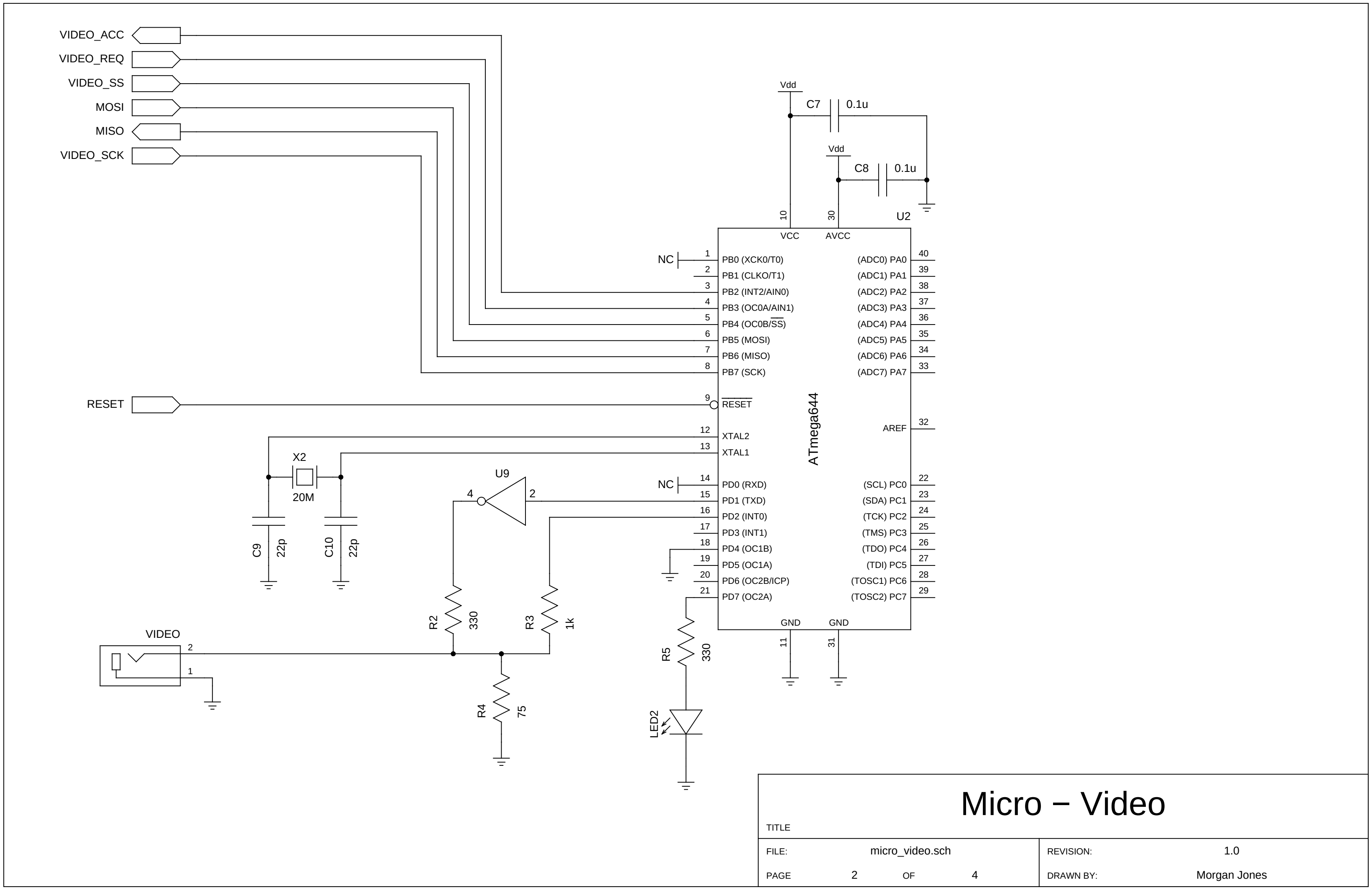
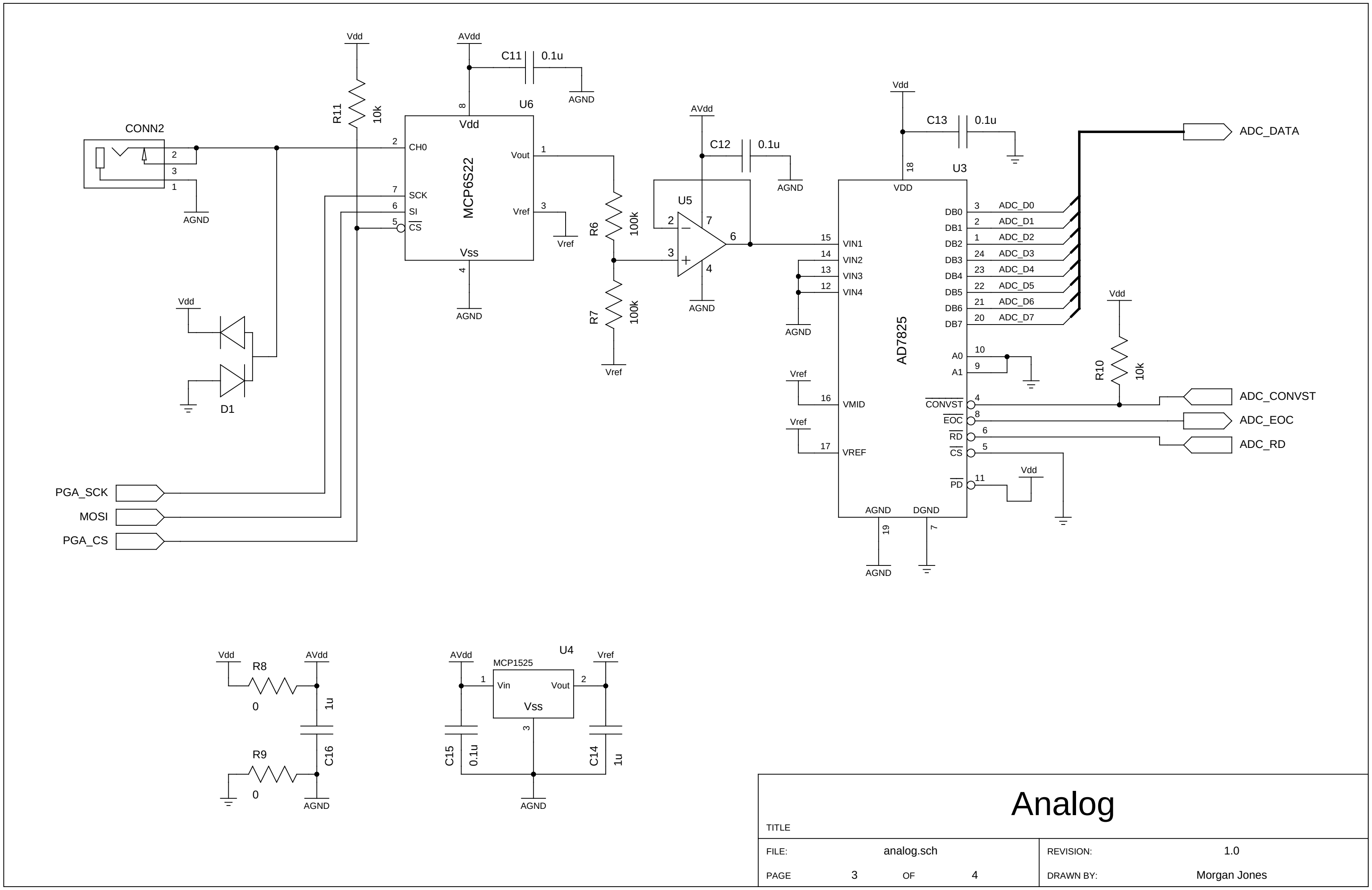


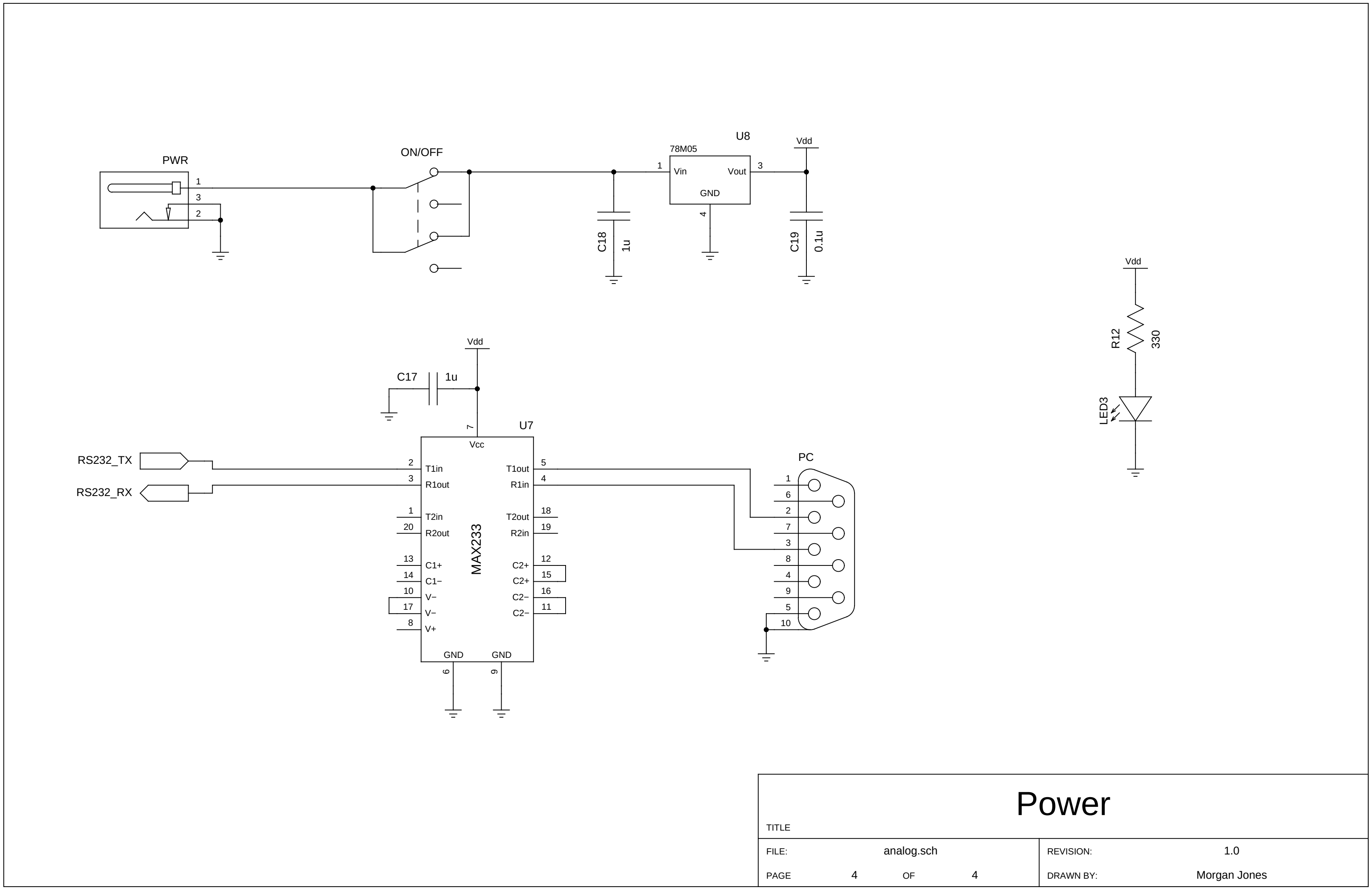
Figure 7: The fully assembled oscilloscope.

A. Schematics

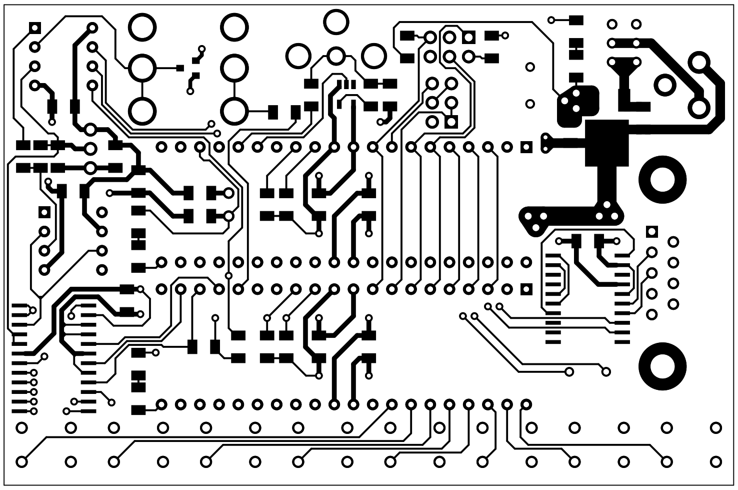



VIDEO_ACC
VIDEO_REQ
VIDEO_SS
MOSI
MISO
VIDEO_SCK
RESET
Vdd
C7
0.1u
Vdd
C8
0.1u
U2
10
30
VCC
AVCC
NC
1 PB0 (XCK0/T0)
2 PB1 (CLKO/T1)
3 PB2 (INT2/AIN0)
4 PB3 (OC0A/AIN1)
5 PB4 (OC0B/SS)
6 PB5 (MOSI)
7 PB6 (MISO)
8 PB7 (SCK)
9 RESET
12 XTAL2
13 XTAL1
NC
14 PD0 (RXD)
15 PD1 (TXD)
16 PD2 (INT0)
17 PD3 (INT1)
18 PD4 (OC1B)
19 PD5 (OC1A)
20 PD6 (OC2B/ICP)
21 PD7 (OC2A)
40 (ADC0) PA0
39 (ADC1) PA1
38 (ADC2) PA2
37 (ADC3) PA3
36 (ADC4) PA4
35 (ADC5) PA5
34 (ADC6) PA6
33 (ADC7) PA7
32 AREF
22 (SCL) PC0
23 (SDA) PC1
24 (TCK) PC2
25 (TMS) PC3
26 (TDO) PC4
27 (TDI) PC5
28 (TOSC1) PC6
29 (TOSC2) PC7
ATmega644
GND
GND
11
31
X2
20M
C9
22p
C10
22p
U9
4
2
R2
330
R3
1k
R4
75
R5
330
LED2
VIDEO
2
1
TITLE
Micro − Video
FILE:
micro_video.sch
REVISION:
1.0
PAGE
2
OF
4
DRAWN BY:
Morgan Jones

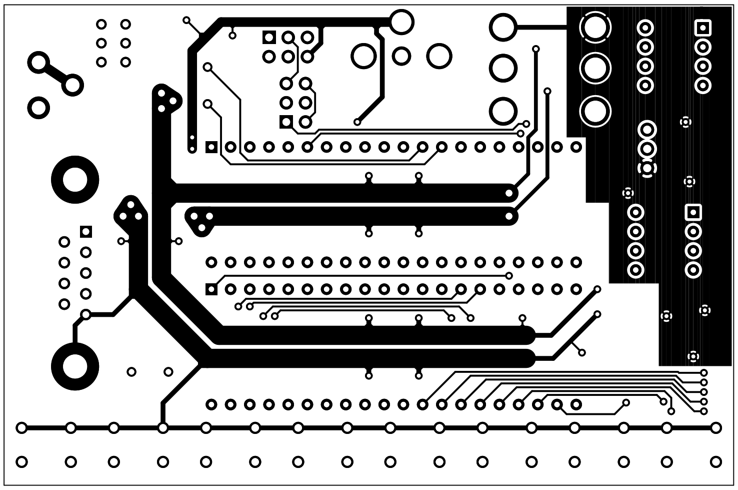

CONN2
2
3
1
AGND
Vdd
R11
10k
AVdd
C11
0.1u
AGND
U6
MCP6S22
Vdd
8
CH0
2
SCK
7
SI
6
CS
5
Vout
1
Vref
3
Vss
4
Vref
AGND
Vdd
D1
R6
100k
R7
100k
Vref
AVdd
C12
0.1u
AGND
U5
2
7
6
3
4
AGND
Vdd
C13
0.1u
18
U3
AD7825
VDD
15 VIN1
14 VIN2
13 VIN3
12 VIN4
AGND
Vref
16 VMID
Vref
17 VREF
AGND 19
DGND 7
AGND
DB0 3 ADC_D0
DB1 2 ADC_D1
DB2 1 ADC_D2
DB3 24 ADC_D3
DB4 23 ADC_D4
DB5 22 ADC_D5
DB6 21 ADC_D6
DB7 20 ADC_D7
ADC_DATA
A0 10
A1 9
Vdd
R10
10k
CONVST 4
EOC 8
RD 6
CS 5
PD 11
Vdd
ADC_CONVST
ADC_EOC
ADC_RD
PGA_SCK
MOSI
PGA_CS
Vdd
R8
0
AVdd
1u
C16
R9
0
AGND
AVdd
MCP1525
U4
Vref
1 Vin
Vout 2
Vss
3
C15
0.1u
C14
1u
AGND
TITLE
Analog
FILE:
analog.sch
REVISION:
1.0
PAGE
3
OF
4
DRAWN BY:
Morgan Jones


PWR
1
3
2
ON/OFF
U8
78M05
1
Vin
Vout
3
GND
4
Vdd
C18
1u
C19
0.1u
Vdd
R12
330
LED3
Vdd
C17
1u
7
Vcc
U7
RS232_TX
RS232_RX
2
T1in
3
R1out
1
T2in
20
R2out
13
C1+
14
C1−
10
V−
17
V−
8
V+
MAX233
5
T1out
4
R1in
18
T2out
19
R2in
12
C2+
15
C2+
16
C2−
11
C2−
GND
GND
6
9
PC
1
6
2
7
3
8
4
9
5
10
TITLE
Power
FILE:
analog.sch
REVISION:
1.0
PAGE
4
OF
4
DRAWN BY:
Morgan Jones

B. PCB Layout



Top copper layer



Bottom copper layer

C. Bill of Materials

Part	Quantity	Digikey P/N
Resistors		
100kΩ	10	RHM100KFRCT-ND
10kΩ	10	RHM10.0KFRCT-ND
1kΩ	10	RHM1.00KFRCT-ND
330Ω	10	RHM330FCT-ND
75Ω	10	RHM75.0FRCT-ND
0Ω	10	RHM0.0ERCT-ND
Capacitors		
1uF	10	PCC2234CT-ND
0.1uF	10	399-1249-1-ND
22pF	10	311-1322-1-ND
Miscellaneous discretes		
Green LED	3	160-1188-1-ND
20MHz crystals	2	X1076-ND
Schottky diode	1	BAT54SFSCT-ND
Logic inverter	1	NC7SZ04M5XCT-ND
Buttons, switches, connectors		
Buttons	8	P12192S-ND
Analog and trigger connectors	2	CP-3502N-ND
RCA video connector	1	CP-1403-ND
Power connector	1	CP-102A-ND
Male pin-header	1	S2011E-36-ND
On/off switch	1	EG1919-ND
Serial connector	1	A35108-ND
Integrated Circuits		
ATmega644 microcontroller	2	ATMEGA644-20PU-ND
MC7805 voltage regulator	1	MC7805BDTRKGOSCT-ND
Sampled Integrated Circuits		
AD7825 ADC	1	
MCP1525 2.5V reference	1	
MCP601 op amp	1	
MCP6S21 PGA	1	
MAX233 RS-232 level shifter	1	

D. Analog Code Listing

```
#include <avr/io.h>

#include <avr/interrupt.h>

#include <util/delay.h>

#include <stdio.h>
#include <string.h>

// button debouncer parameters
#define BUT_NCHECKS     4
#define BUT_IMASK       0x03

// sampling speed grades
#define S_LIGHT         1
#define S_RIDICULOUS    2
#define S_LUDICROUS     3

// buttons
uint8_t but_index;
uint8_t but_state[BUT_NCHECKS];
uint8_t but_count;
uint8_t but_repeat;

// array of samples
uint8_t samples[200];

// sample acquisition
volatile uint8_t samp_flag;
volatile uint8_t sample_new;
volatile uint8_t sample_old;
uint8_t samp_index;
uint8_t samp_acq_index;

// time division
uint8_t samp_speed_grade;
uint8_t samp_rate;
uint16_t samp_rate_table[13] =
{
        62500, // 5s, fosc/8
        25000, // 2s
        12500, // 1s
        6250, // 0.5s
        20000, // 0.2s, fosc/1
        10000, // 0.1s
        5000, // 50ms
        2000, // 20ms
        1000, // 10ms
        500, // 5ms
        200, // 2ms
        100, // 1ms
        50 // 0.5ms
};

// voltage division
uint8_t volt_setting;
```

```
// capture state
uint8_t running;
uint8_t triggered;
uint8_t single_seq;
volatile uint8_t capt_count;

// parameter being editted
uint8_t edit_mode;

// trigger and cursor
uint8_t trig_lvl;
uint8_t x_curs;

// voltage divisions in text
char *volt_text[] =
{
        "5.0V ",
        "2.5V ",
        "1.25V",
        "1.0V ",
        "625mV",
        "500mV",
        "313mV",
        "156mV"
};

// voltage divisions in microvolts
int32_t volt_uv[] =
{
        5000000,
        2500000,
        1250000,
        1000000,
        625000,
        500000,
        312500,
        156250
};

// time divisions in text
char *time_text[] =
{
        "5.0s ",
        "2.0s ",
        "1.0s ",
        "500ms",
        "200ms",
        "100ms",
        "50ms ",
        "20ms ",
        "10ms ",
        "5.0ms",
        "2.0ms",
        "1.0ms",
        "0.5ms"
};

// time divisions in microseconds
```

```
int32_t time_us[] =
{
        5000000,
        2000000,
        1000000,
        500000,
        200000,
        100000,
        50000,
        20000,
        10000,
        5000,
        2000,
        1000,
        500
};

// header and footer
char header[26] = ">H:       V:            R\0";
char footer[26] = " Ct:         CV:         \0";

// footer trigger and cursor readouts
char footer_trig[25] = "T:                      \0";
char footer_curs[25] = "Ct:         CV:         \0";

/*
 * INT1: Reads sample from ADC
 */
ISR(INT1_vect)
{
    sample_old = sample_new;
    PORTD &= ~(_BV(PD5));
    sample_new = PINC;
    PORTD |= _BV(PD5);

    samp_flag = 1;
}

/*
 * TIMER2_COMPA: Button debouncer, also generates ~10ms time base
 */
ISR(TIMER2_COMPA_vect, ISR_NOBLOCK)
{
    but_state[but_index++ & BUT_IMASK] = PINA;
    if (capt_count > 0) capt_count--;
    if (but_repeat > 0) but_repeat--;
}

/*
 * TIMER1_COMPA: Starts an ADC conversion
 */
ISR(TIMER1_COMPA_vect, ISR_NAKED)
{
    // Start conversion
    asm volatile(
            "cbi 0x0B, 4"       "\n\t"
            "sbi 0x0B, 4"       "\n\t"
            "reti"
            : :
```

```
            );
}

// function prototypes
void verify();
void initialize();

void sendPGA(uint8_t, uint8_t);

void sendVideoCmnd(uint8_t);
void sendVideoSamp(uint8_t);
void sendHeader();
void sendFooter();

void setMode(uint8_t);
void setTriggerReadout(uint8_t);
void setCursorReadout(uint8_t);
void setWindow(uint8_t,uint8_t);

inline void captureLight();
inline void captureRidiculous();
inline void captureLudicrous();

int main()
{
    uint8_t i;
    uint8_t but = 0, last_but = 0, but_flag = 0, rel_flag = 0;

    verify();
    initialize();

    sei();

    sendVideoCmnd(0xA0);
    setMode(0);
    setTriggerReadout(trig_lvl);
    setWindow(samp_rate, volt_setting);

    while (1)
    {
        last_but = but;
        but = 0xFF;
        for (i = 0; i < BUT_NCHECKS; i++)
            but &= ~(but_state[i]);
        but_flag = but & ~last_but;
        rel_flag = ~but & last_but;

        // UP
        if (but_flag & 0x01)
        {
            if (edit_mode == 0 && samp_rate < 12)
            {
                // increase time division
                setWindow(++samp_rate, volt_setting);
            }
            else if (edit_mode == 1 && volt_setting < 7)
            {
                // increase voltage division
                setWindow(samp_rate, ++volt_setting);
            }
            else if (edit_mode == 2 && x_curs < 199)
            {
                // increase cursor coordinate
                sendVideoCmnd(0xA3);
                sendVideoCmnd(++x_curs);
                but_repeat = 50;
            }
            else if (edit_mode == 3 && trig_lvl < 255)
            {
                // increase trigger level
                sendVideoCmnd(0xA4);
                sendVideoSamp(++trig_lvl);
                setTriggerReadout(trig_lvl);
                sendFooter();
                but_repeat = 50;
            }
        }
        else if (but & 0x01 && but_repeat == 0)
        {
            if (edit_mode == 2 && x_curs < 199)
            {
                // continue increasing cursor coordinate
                sendVideoCmnd(0xA3);
                sendVideoCmnd(++x_curs);
                but_repeat = 5;
            }
            else if (edit_mode == 3 && trig_lvl < 255)
            {
                // continue increasing trigger level
                sendVideoCmnd(0xA4);
                sendVideoSamp(++trig_lvl);
                setTriggerReadout(trig_lvl);
                sendFooter();
                but_repeat = 5;
            }
        }
        else if (rel_flag & 0x01)
        {
            if (edit_mode == 2)
            {
                // update cursor readout
                setCursorReadout(x_curs);
                sendFooter();
            }
            else if (edit_mode == 3)
            {
                // erase trigger indicator on video
                sendVideoCmnd(0xA4);
                sendVideoCmnd(255);
            }
        }

        // DOWN
        if (but_flag & 0x02)
        {
            if (edit_mode == 0 && samp_rate > 0)
            {
                // decrease time division
                setWindow(--samp_rate, volt_setting);
            }
            else if (edit_mode == 1 && volt_setting > 0)
            {
                // decrease voltage division
                setWindow(samp_rate, --volt_setting);
            }
            else if (edit_mode == 2 && x_curs > 0)
            {
                // decrease cursor coordinate
                sendVideoCmnd(0xA3);
                sendVideoCmnd(--x_curs);
                but_repeat = 50;
            }
            else if (edit_mode == 3 && trig_lvl > 0)
            {
                // decrease trigger level
                sendVideoCmnd(0xA4);
                sendVideoSamp(--trig_lvl);
                setTriggerReadout(trig_lvl);
                sendFooter();
                but_repeat = 50;
            }
        }
        else if (but & 0x02 && but_repeat == 0)
        {
            if (edit_mode == 2 && x_curs > 0)
            {
                // continue decreasing cursor coordinate
                sendVideoCmnd(0xA3);
                sendVideoCmnd(--x_curs);
                but_repeat = 5;
            }
            else if (edit_mode == 3 && trig_lvl > 0)
            {
                // continue decreasing trigger level
                sendVideoCmnd(0xA4);
                sendVideoSamp(--trig_lvl);
                setTriggerReadout(trig_lvl);
                sendFooter();
                but_repeat = 5;
            }
        }
        else if (rel_flag & 0x02)
        {
            if (edit_mode == 2)
            {
                // update cursor readout
                setCursorReadout(x_curs);
                sendFooter();
            }
            else if (edit_mode == 3)
            {
                // erase trigger indicator on video
                sendVideoCmnd(0xA4);
                sendVideoCmnd(255);
            }
        }

        // CRS
        if (but_flag & 0x04)
        {
            if (!running)
                setMode(2);
        }

        // VOLT
        if (but_flag & 0x08)
        {
            setMode(1);
        }

        // TIME
        if (but_flag & 0x10)
        {
            setMode(0);
        }

        // TRIG
        if (but_flag & 0x20)
        {
            setMode(3);
        }

        // SEQ
        if (but_flag & 0x40)
        {
            if (!running)
            {
                // write SEQ in header
                header[20] = ’S’;
                header[21] = ’E’;
                header[22] = ’Q’;
                setWindow(samp_rate, volt_setting);
                single_seq = 1;
            }
        }

        // RUN
        if (but_flag & 0x80)
        {
            if (running)
            {
                // stop running
                running = 0;
                header[24] = ’S’;
                sendHeader();
                sendFooter();
            }
            else
            {
                // start running
                running = 1;
                header[24] = ’R’;
                sendVideoCmnd(0xA3);
                sendVideoCmnd(255);
                if (edit_mode == 2)
                    setMode(3);
                if (single_seq)
                {
                    header[20] = ' ';
                    header[21] = ' ';
                    header[22] = ' ';
                    single_seq = 0;
                }
                setTriggerReadout(trig_lvl);
                setWindow(samp_rate, volt_setting);
            }
        }

        // capture data
        if (running || single_seq)
        {
            if (samp_speed_grade == S_LIGHT)
                captureLight();
            else if (samp_speed_grade == S_RIDICULOUS)
                captureRidiculous();
            else if (samp_speed_grade == S_LUDICROUS)
                captureLudicrous();

            // single sequence complete, erase SEQ
            if (!running && !single_seq)
            {
                header[20] = ' ';
                header[21] = ' ';
                header[22] = ' ';
                sendHeader();
            }
        }
    }

    return 0;
}

/*
 * SENDPGA: Send instruction inst and value val to PGA
 */
void sendPGA(uint8_t inst, uint8_t val)
{
    // set CS low
    PORTB &= ~(_BV(PB0));

    // send instruction, wait for transmission to complete
    SPDR = inst;
    while (!(SPSR & 0x80)) ;

    // send value, wait for transmission to complete
    SPDR = val;
    while (!(SPSR & 0x80)) ;

    // set CS high
    PORTB |= _BV(PB0);
}

/*
 * SENDVIDEO: Send a byte to the video microcontroller
 */
```

```
inline void sendVideo(uint8_t c)
{
    // wait for ACC high
    while (!(PINB & _BV(PINB2))) ;

    // set REQ low
    PORTB &= ~(_BV(PB3));

    // wait for ACC low
    while (PINB & _BV(PINB2)) ;

    // set CS low
    PORTB &= ~(_BV(PB4));

    // send byte, wait for transmission to complete
    SPDR = c;
    while (!(SPSR & 0x80)) ;

    // set CS high
    PORTB |= _BV(PB4);

    // set REQ high
    PORTB |= _BV(PB3);
}

/*
 * SENDVIDEOCMND: Send a command to the video microcontroller
 */
void sendVideoCmnd(uint8_t c)
{
    sendVideo(c);
}

/*
 * SENDVIDEOSAMP: Send a sample to the video microcontroller
 */
void sendVideoSamp(uint8_t s)
{
    sendVideo(127-(uint8_t)((((int8_t)(s-128))>>1) + 64));
}

/*
 * SENDHEADER: Send the header string to the video microcontroller
 */
void sendHeader()
{
    uint8_t i;

    sendVideoCmnd(0xA1);
    for (i = 0; i < 25; i++)
        sendVideoCmnd(header[i]);
    sendVideoCmnd(0xA0);
}

/*
 * SENDFOOTER: Send the footer string to the video microcontroller
 */
void sendFooter()
{
    uint8_t i;

    sendVideoCmnd(0xA2);
    for (i = 0; i < 25; i++)
        sendVideoCmnd(footer[i]);
    sendVideoCmnd(0xA0);
}

/*
 * SETMODE: Sets the current edit mode to m
 */
void setMode(uint8_t m)
{
    // erase mode indicators
    header[0] = ' ';
    header[9] = ' ';
    footer[0] = ' ';

    // place mode indicator appropriately
    if (m == 0)
        header[0] = '>';
    else if (m == 1)
        header[9] = '>';
    else if (m == 2)
    {
        setCursorReadout(x_curs);
        footer[0] = '>';
    }
    else if (m == 3)
    {
        setTriggerReadout(trig_lvl);
        footer[0] = '>';
    }

    // send header & footer
    sendHeader();
    sendFooter();

    // turn on cursor if entering cursor mode, turn of if exiting
    if (edit_mode == 2 && m != 2)
    {
        sendVideoCmnd(0xA3);
        sendVideoCmnd(255);
    }
    else if (m == 2)
    {
        sendVideoCmnd(0xA3);
        sendVideoCmnd(x_curs);
    }

    // set current mode
    edit_mode = m;
}

/*
 * SETTRIGGERREADOUT: Set the footer string to have the trigger level t
 */
void setTriggerReadout(uint8_t t)
{
    int32_t y = (int32_t)((int8_t)(t-128));
    int32_t v;
    int16_t l,r;
    uint8_t neg = 0;

    strcpy(footer+1, footer_trig);

    v = (volt_uv[volt_setting]*y)>>8;
    if (v < 0)
    {
        neg = 1;
        v = -v;
    }

    if (volt_setting < 4)
    {
        l = (int16_t)(v/1000000);
        r = (int16_t)((v%1000000)/1000);
        snprintf(footer+4, 8, "%d.%03dV ", l, r);
    }
    else
    {
        l = (int16_t)(v/1000);
        r = (int16_t)((v%1000)/100);
        snprintf(footer+4, 8, "%3d.%dmV", l, r);
    }

    footer[3] = neg ? '-' : ' ';
}

/*
 * SETCURSORREADOUT: Set the footer string to have the cursor data for
 *      position x
 */
void setCursorReadout(uint8_t x)
{
    int32_t y = (int32_t)((int8_t)(samples[x]-128));//<<8;
    int32_t v;
    int16_t l,r;
    uint8_t neg = 0;

    strcpy(footer+1, footer_curs);

    v = (time_us[samp_rate]*x)/200;

    if (samp_rate < 3)
    {
        l = (int16_t)(v/1000000);
        r = (int16_t)((v%1000000)/1000);
        snprintf(footer+4, 9, "%d.%03ds ", l, r);
    }
    else if (samp_rate < 6)
    {
        l = (int16_t)(v/1000);
        r = (int16_t)((v%1000)/100);
        snprintf(footer+4, 9, "%3d.%dms", l, r);
    }
    else if (samp_rate < 9)
    {
        l = (int16_t)(v/1000);
        r = (int16_t)((v%1000)/10);
        snprintf(footer+4, 9, "%2d.%02dms", l, r);
    }
    else if (samp_rate < 12)
    {
        l = (int16_t)(v/1000);
        r = (int16_t)(v%1000);
        snprintf(footer+4, 9, "%d.%03dms", l, r);
    }
    else
    {
        l = (int16_t)(v);
        snprintf(footer+4, 9, "%3dus  ", l);
    }

    v = (volt_uv[volt_setting]*y)>>8;
    if (v < 0)
    {
        neg = 1;
        v = -v;
    }

    if (volt_setting < 4)
    {
        l = (int16_t)(v/1000000);
        r = (int16_t)((v%1000000)/1000);
        snprintf(footer+17, 8, "%d.%03dV ", l, r);
    }
    else
    {
        l = (int16_t)(v/1000);
        r = (int16_t)((v%1000)/100);
        snprintf(footer+17, 8, "%3d.%dmV", l, r);
    }

    footer[16] = neg ? ’-’ : ’ ’;
}

/*
 * SETWINDOW: Set the graphing window, rate is sample rate, volt is voltage
 *      division
 */
void setWindow(uint8_t rate, uint8_t volt)
{
    // turn off and reset timer 1
    TCCR1B &= ~(_BV(CS12) | _BV(CS11) | _BV(CS10));
    TIMSK1 &= ~(_BV(OCIE1A));
    TIFR1 |= _BV(OCF1A);
    TCNT1 = 0;

    // set sample rate
    OCR1A = samp_rate_table[rate];

    // disable external interrupt 1
    EIMSK &= ~(_BV(INT1));
    EIFR |= _BV(INTF1);

    // re-initialize various variables
    triggered = 0;
    samp_flag = 0;
    samp_acq_index = 0;

    // update header
    strcpy(header+3, time_text[rate]);
    strcpy(header+12, volt_text[volt]);

    // send header and footer to video
    sendHeader();
    sendFooter();

    // send gain to PGA
    sendPGA(0x40, volt);

    if (rate < 4)
    {
        // Light speed!
        sendVideoCmnd(0x90);
        samp_index = 0;
        samp_speed_grade = S_LIGHT;
        TCCR1B |= _BV(CS11);    // turn on timer 1 with clock fosc/8
        TIMSK1 |= _BV(OCIE1A);  // turn on OC1A interrupt
        EIMSK |= _BV(INT1);     // turn on external interrupt 1
    }
    else if (rate < 9)
    {
        // Ridiculous speed!
        sendVideoCmnd(0x80);
        samp_index = 200;
        sample_old = 255;
        samp_speed_grade = S_RIDICULOUS;
        TCCR1B |= _BV(CS10);    // turn on timer 1 with clock fosc/1
        TIMSK1 |= _BV(OCIE1A);  // turn on OC1A interrupt
        EIMSK |= _BV(INT1);     // turn on external interrupt 1
    }
    else
    {
        // LUDICROUS SPEED!!!
        sendVideoCmnd(0x80);
        samp_index = 200;
        samp_speed_grade = S_LUDICROUS;
        // leave timer 1 off
        // leave OC1A interrupt off
        // leave external interrupt 1 off
        capt_count = 0; // start sampling immediately
    }
}

/*
 * CAPTURELIGHT: Capture samples at the slow speed grade, immediately
 *      display samples
 */
inline void captureLight()
{
    if (samp_flag)
    {
        samp_flag = 0;
        // store sample
        samples[samp_index++] = sample_new;

        // send sample
        sendVideoSamp(sample_new);

        if (samp_index == 200)
        {
            sendVideoCmnd(0x80);
            samp_index = 0;
            single_seq = 0;
        }
    }
}

/*
 * CAPTURERIDICULOUS: Capture samples at the medium speed grade, capture
 *      200 samples after trigger and display after all captured
 */
inline void captureRidiculous()
{
    uint8_t curr, prev;

    if (samp_flag && capt_count == 0)
    {
        samp_flag = 0;

        curr = sample_new;
        prev = sample_old;

        // check for trigger
        if (prev < trig_lvl && curr >= trig_lvl)
            triggered = 1;

        if (triggered)
        {
            // store sample
            samples[samp_acq_index++] = curr;

            if (samp_acq_index == 200)
            {
                samp_acq_index = 0;
                triggered = 0;

                // start sample transmission
                sendVideoCmnd(0x80);
                samp_index = 0;

                // wait for 500ms before next read cycle
                capt_count = 50;
            }
        }
    }

    if (samp_index < 200)
    {
        // send sample
        sendVideoSamp(samples[samp_index++]);
        if (samp_index == 200)
```

```
            single_seq = 0;
    }
}

/*
 * CAPTURELUDICROUS: Capture samples at the medium speed grade, capture
 *      200 samples after trigger and display after all captured
 */
inline void captureLudicrous()
{
    uint8_t curr_samp = 255, prev_samp;
    uint8_t* samp_ptr;
    uint8_t timeout = 200;

    if (capt_count == 0)
    {
        PIND = _BV(PIND7);

        samp_ptr = samples;
        triggered = 0;

        // temporarily shut off button debouncer
        TCCR2B &= ~(_BV(CS22) | _BV(CS21) | _BV(CS20));

        EIFR |= _BV(INTF1);      // make sure INT1 flag is clear
        TCNT1 = 0;               // reset counter 1
        TIFR1 |= _BV(OCF1A);     // make sure OC1A flag is clear
        TCCR1B |= _BV(CS10);     // start timer 1
        TIMSK1 |= _BV(OCIE1A);   // turn on OC1A interrupt

        // wait for trigger, then acquire 200 samples
        while (samp_ptr < samples+200 && timeout > 0)
        {
            prev_samp = curr_samp;
            while (!(EIFR & _BV(INTF1))) ;
            EIFR |= _BV(INTF1);

            PORTD &= ~(_BV(PD5));
            curr_samp = PINC;
            PORTD |= _BV(PD5);

            if (prev_samp < trig_lvl && curr_samp >= trig_lvl)
                triggered = 1;

            if (triggered)
                *(samp_ptr++) = curr_samp;
            else
                timeout--;
        }

        TIMSK1 &= ~(_BV(OCIE1A));   // turn off OC1A interrupt
        TCCR1B &= ~(_BV(CS10));     // stop timer 1

        // restart button debouncer
        TCCR2B |= _BV(CS22) | _BV(CS21) | _BV(CS20);

        if (timeout > 0)
        {
            // start sample transmission
            sendVideoCmnd(0x80);
            samp_index = 0;

            // wait for 500ms before next read cycle
            capt_count = 50;
        }
        else
        {
            // timed out, wait for 500ms before trying again
            capt_count = 50;
        }
    }

    if (samp_index < 200)
    {
        // send sample
        sendVideoSamp(samples[samp_index++]);
        if (samp_index == 200)
            single_seq = 0;
    }
}

/*
 * INITIALIZE: Set up the microcontroller registers and initialize global
 *      variables.
 */
void initialize()
{
    uint8_t i;

    // PA7: RUN         in
    // PA6: SEQ         in
    // PA5: TRIG        in
    // PA4: TIME        in
    // PA3: VOLT        in
    // PA2: CRS         in
    // PA1: DOWN        in
    // PA0: UP          in
    DDRA = 0;
    PORTA = 0xFF; // internal pull-ups

    // PB7: SCK         out
    // PB6: MISO        in
    // PB5: MOSI        out
    // PB4: VIDEO CS    out
    // PB3: VIDEO REQ   out
    // PB2: VIDEO ACC   in
    // PB1: (unused)
    // PB0: PGA CS      out
    PORTB = _BV(PB4) | _BV(PB3) | _BV(PB0);
    DDRB = _BV(DDB7) | _BV(DDB5) | _BV(DDB4) | _BV(DDB3) | _BV(DDB0);

    // PC7-0: ADC data  in
    DDRC = 0;

    // PD7: LED         out
    // PD6: (unused)
    // PD5: ADC RD      out
    // PD4: ADC CONVST  out
    // PD3: ADC EOC      in
    // PD2: EXT TRIG     in
    // PD1: RS232 TX
    // PD0: RS232 RX
    PORTD = _BV(PD5) | _BV(PD4);
    DDRD = _BV(DDD7) | _BV(DDD5) | _BV(DDD4);

    _delay_ms(5);

    // INT1: ADC EOC (falling edge)
    EICRA = _BV(ISC11);
    // don't enable external interrupt 1 yet

    // SPI: MSB first, master, mode 00, fosc/4 (5 MHz)
    SPCR = _BV(SPE) | _BV(MSTR);

    // TIMER 1 (sample rate): OC1* disconnected, CTC mode
    TCCR1B = _BV(WGM12);    // leave it off for now

    // TIMER 2 (buttons): OC2* disconnected, CTC mode, fosc/1024 (~19.5kHz),
    //        OC2A interrupt enabled
    TCCR2A = _BV(WGM21);
    TCCR2B = _BV(CS22) | _BV(CS21) | _BV(CS20);
    OCR2A = 195; // ~10 ms time base
    TIMSK2 = _BV(OCIE2A);

    // Button state
    for (i = 0; i < BUT_NCHECKS; i++)
        but_state[i] = 0xFF;
    but_index = 0;

    // Samples
    samp_flag = 0;
    samp_index = 0;
    samp_rate = 0;
    samp_acq_index = 0;

    running = 1;

    triggered = 0;

    single_seq = 0;

    // trigger level at 0V
    trig_lvl = 128;

    edit_mode = 0;
    volt_setting = 0;

    // cursor in middle
    x_curs = 100;
}

/*
 * VERIFY: Check to make sure the code is running on the correct
 *      microcontroller. If not, flash a warning on the LED forever.
 */
void verify()
{
    uint8_t i;

    // if pin 4 on port D is high, everything's fine!
    if (PIND & _BV(PIND4))
        return;

    // we're executing on the wrong microcontroller!
    DDRD |= _BV(DDD7);
    while (1)
    {
        // flash the LED forever
        PORTD |= _BV(PD7);
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD &= ~(_BV(PD7));
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD |= _BV(PD7);
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD &= ~(_BV(PD7));
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD |= _BV(PD7);
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD &= ~(_BV(PD7));
        for (i = 0; i < 100; i++)
            _delay_ms(10);
    }
}
```

E. Video Code Listing

```
#include <avr/io.h>

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

#include <util/delay.h>

// video timing
#define LINE_TIME 1271
#define SLEEP_TIME 1250

// screen layout
#define V_TOP       55
#define V_HEADER    55
#define V_HEADER_SP 64
#define V_HEADER_R  65
#define V_GRAPH     66
#define V_FOOTER_R  194
#define V_FOOTER_SP 195
#define V_FOOTER    196
#define V_BOTTOM    205

// screen sections
#define S_BLANK     0
#define S_TEXT      1
#define S_GRAPH     2
#define S_RULE      3

// video generation
uint8_t sync_on, sync_off;
uint16_t video_line;
uint8_t char_line;
uint16_t graph_offset;
uint8_t section;
uint8_t* text;

// cursor and trigger
uint8_t x_cursor;
uint8_t y_trigger;

// header and footer
uint8_t header[26] = "                         \0";
uint8_t footer[26] = "                         \0";

// bitmask for drawing points
uint8_t bitmask[8] PROGMEM = {0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};

// graph raster
uint8_t graph[3200];

// array of samples
uint8_t samples[200];

/*
 * TIMER2_COMPB: Puts the microcontroller to sleep before new scanline
 */
ISR(TIMER1_COMPB_vect, ISR_NAKED)
{
    sei();
    sleep_cpu();
    reti();
}

/*
 * TIMER1_COMPA: Generates sync pulses and scanlines.  Must be entered from
 *      sleep mode for consistent timing
 */
ISR(TIMER1_COMPA_vect)
{
    uint8_t out, x;

    // start the horizontal sync pulse
    PORTD = (PORTD & ~(_BV(PORTD2))) | sync_on;

    // begin inverted (vertical) sync after line 247
    if (video_line==247)
    {
        sync_on = _BV(PORTD2);
        sync_off = 0;
    }

    // back to regular sync after line 250
    if (video_line==250)
    {
        sync_on = 0;
        sync_off = _BV(PORTD2);
    }

    // adjust to make 5 us pulses
    _delay_us(4);

    // end sync pulse
    PORTD = (PORTD & ~(_BV(PORTD2))) | sync_off;

    // don't send data to the screen for non-visible portions of the screen,
    // this allows more time for the main loop to execute
    if (video_line < V_BOTTOM && video_line >= V_TOP)
    {
        // adjust to center the image
        _delay_us(7);

        // send an initial blank byte
        if (section != S_BLANK)
            UDR0 = 0xFF;

        // send data data depending on screen section
        switch (section)
        {
        case S_TEXT:
            // draw character data
            for (x = 0; x < 25; x++)
            {
                while (!(UCSR0A & _BV(UDRE0))) ;
                asm volatile(
                    "ld r30, %a1"               "\n\t"
                    "ldi r31, hi8(chartable)"   "\n\t"
                    "add r31, %2"               "\n\t"
                    "lpm %0, Z"
                    : "=r" (out)
                    : "e" (text), "r" (char_line)
                    : "r30", "r31"
                    );
                UDR0 = ~(out);
                text++;
            }
            break;
        case S_GRAPH:
            // draw graph data, or white line if at trigger
            if (video_line-V_GRAPH == y_trigger)
            {
                for (x = 0; x < 25; x++)
                {
                    while (!(UCSR0A & _BV(UDRE0))) ;
                    UDR0 = 0;
                }

            }
            else
            {
                for (x = 0; x < 25; x++)
                {
                    while (!(UCSR0A & _BV(UDRE0))) ;
                    UDR0 = ~(graph[graph_offset++]);
                }
            }
            break;
        case S_RULE:
            // draw a white line across the entire screen
            for (x = 0; x < 25; x++)
            {
                while (!(UCSR0A & _BV(UDRE0))) ;
                UDR0 = 0x00;
            }
            break;
        case S_BLANK:
            // do nothing
            break;
        }
    }

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

    // set the screen section for next line
    if (video_line >= V_TOP && video_line < V_BOTTOM)
    {
        if (video_line < V_HEADER_SP)
        {
            section = S_TEXT;
            text = header;
            char_line = video_line-V_HEADER;
        }
        else if (video_line < V_HEADER_R)
            section = S_BLANK;
        else if (video_line < V_GRAPH)
            section = S_RULE;
        else if (video_line < V_FOOTER_R)
        {
            section = S_GRAPH;
            graph_offset = (video_line-V_GRAPH)*25;
        }
        else if (video_line < V_FOOTER_SP)
            section = S_RULE;
        else if (video_line < V_FOOTER)
            section = S_BLANK;
        else
        {
            section = S_TEXT;
            text = footer;
            char_line = video_line-V_FOOTER;
        }
    }
    else
    {
        section = S_BLANK;
    }
}

// function prototypes
void verify();
void initialize();
void drawPoint(uint8_t x, uint8_t y, uint8_t c);
void drawLine(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, uint8_t c);
void drawCursor(uint8_t x, uint8_t c);

int main(void)
{
    uint8_t x = 0;
    uint8_t in;
    uint8_t mode = 0;
    uint8_t char_count = 0;

    verify();
    initialize();

    // turn on interrupts
    sei();

    while (1)
    {
        // wait for REQ to be pulled low, set ACC low
        while (PINB & _BV(PINB3)) ;
        PORTB &= ~(_BV(PORTB2));

        // wait for reception to be complete, store byte
        while (!(SPSR & _BV(SPIF))) ;
        in = SPDR;

        // wait for REQ to be pulled high, set ACC high
        while (!(PINB & _BV(PINB3))) ;
        PORTB |= _BV(PORTB2);
        // bling LED
        PIND = _BV(PIND7);

        if (in & 0x80 && mode != 3 && mode != 4)
        {
            // command received

            switch (in & 0xF0)
            {
            case 0x80:
                // reset x-counter
                x = 0;
                break;
            case 0x90:
                // clear screen
                for (x = 0; x < 200; x++)
                {
                    if (x < 199)
                        drawLine(x, samples[x], x+1, samples[x+1], 0);

                    samples[x] = 0;
                }

                // redraw cursor, if its visible
                if (x_cursor != 255)
                    drawCursor(x_cursor, 1);

                // reset x-counter
                x = 0;
                break;
            case 0xA0:
                // change mode
                if ((in & 0x0F) == 0)
                    mode = 0;
                else if ((in & 0x0F) == 1)
                {
                    mode = 1;
                    char_count = 0;
                }
                else if ((in & 0x0F) == 2)
                {
                    mode = 2;
                    char_count = 0;
                }
                else if ((in & 0x0F) == 3)
                    mode = 3;
                else if ((in & 0x0F) == 4)
                    mode = 4;
                break;
            default:
                break;
            }
        }
        else if (mode == 1)
        {
            // store character to header
            if (char_count < 25)
                header[char_count++] = in;
        }
        else if (mode == 2)
        {
            // store character to footer
            if (char_count < 25)
                footer[char_count++] = in;
        }
        else if (mode == 3)
        {
            // erase previous cursor, if it was visible
            if (x_cursor != 255)
            {
                drawCursor(x_cursor, 0);

                // fix the waveform
                if (x_cursor > 0)
                {
                    drawLine(x_cursor-1, samples[x_cursor-1],
                            x_cursor, samples[x_cursor], 1);
                }
                if (x_cursor < 199)
                {
                    drawLine(x_cursor, samples[x_cursor], x_cursor+1,
                            samples[x_cursor+1], 1);
                }
            }

            // store new cursor
            x_cursor = in;

            // draw new cursor, if it is visible
            if (x_cursor != 255)
                drawCursor(x_cursor, 1);

            // Always return to mode 0!
            mode = 0;
        }
        else if (mode == 4)
        {
            // store new trigger
            y_trigger = in;

            // Always return to mode 0!
            mode = 0;
        }
        else
        {
            if (x < 200)
            {
                // erase old line
                if (x < 199)
                    drawLine(x, samples[x], x+1, samples[x+1], 0);

                // store new sample
                samples[x] = in & 0x7F;

                // draw new line
                if (x > 0)
                    drawLine(x-1, samples[x-1], x, samples[x], 1);
                // redraw cursor
                if (x_cursor != 255)
                    drawCursor(x_cursor, 1);

                // increment x-counter
                x++;
            }
        }
    }

    return 0;
}

/*
 * DRAWPOINT: Draws a point at (x,y) color c: 0=black, 1=white, 2=invert
 */
void drawPoint(uint8_t x, uint8_t y, uint8_t c)
{
    asm volatile(
            // i = (x>>3) + (y*25)
            // the byte with the pixel in it

            // get y
            "mov r16, %1"           "\n\t"

            // mult y by 25
            "ldi r17, 25"           "\n\t"
            "mul r16, r17"          "\n\t"

            // get x
            "mov r16, %0"           "\n\t"

            // divide x by 8
            "lsr r16"               "\n\t"
            "lsr r16"               "\n\t"
            "lsr r16"               "\n\t"

            // add in x/8
            "clr r17"               "\n\t"
            "add r16, r0"           "\n\t"
            "adc r17, r1"           "\n\t"

            "clr __zero_reg__"      "\n\t"

            // move screen to X (r27:r26)
            "mov r26, %A3"          "\n\t"
            "mov r27, %B3"          "\n\t"
            "add r26, r16"          "\n\t"
            "adc r27, r17"          "\n\t"

            // get screen byte
            "ld r5, X"              "\n\t"

            // form x & 7
            "mov r30, %0"           "\n\t"
            "andi r30, 0x07"        "\n\t"

            // get bit mask
            "clr r31"               "\n\t"
            "add r30, %A4"          "\n\t"
            "adc r31, %B4"          "\n\t"
            "lpm r6, Z"             "\n\t"

            // if (v4==1) graph[i] = v2 | v3 ;
            // if (v4==0) graph[i] = v2 & ~v3;
            // if (v4==2) graph[i] = v2 ^ v3 ;
            "cpi %2, 1"             "\n\t"
            "brne 1f"               "\n\t"
            "or  r5, r6"            "\n\t"
            "1:"                    "\n\t"
            "cpi %2, 0"             "\n\t"
            "brne 2f"               "\n\t"
            "com r6"                "\n\t"
            "and r5, r6"            "\n\t"
            "2:"                    "\n\t"
            "cpi %2, 2"             "\n\t"
            "brne 3f"               "\n\t"
            "eor r5, r6"            "\n\t"
            "3:"                    "\n\t"

            // write the byte back to the screen
            "st X, r5"
            :
            : "r" (x), "r" (y), "d" (c), "r" (graph), "r" (bitmask)
            : "r5", "r6", "r16", "r17", "r26", "r27", "r30", "r31"
    );
}

/*
 * DRAWLINE: draw a line from (x1,y1) to (x2,y2), color c: 0=black, 1=white,
 *      2=invert
 *
 * Code is from David Rodgers,
 * "Procedural Elements of Computer Graphics",1985
 */
void drawLine(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, uint8_t c)
{
    int16_t e;
    uint8_t dx,dy,j, temp;
    int8_t s1,s2;
    uint8_t x,y, xchange;

    x = x1;
    y = y1;
    dx = (x1 < x2) ? x2-x1 : x1-x2;
    dy = (y1 < y2) ? y2-y1 : y1-y2;
    s1 = (x1 == x2) ? 0 : ((x1 < x2) ? 1 : -1);
    s2 = (y1 == y2) ? 0 : ((y1 < y2) ? 1 : -1);
    xchange = 0;
    if (dy>dx)
    {
        temp = dx;
        dx = dy;
        dy = temp;
        xchange = 1;
    }
    e = ((int16_t)dy<<1) - dx;
    for (j=0; j<=dx; j++)
    {
        drawPoint(x,y,c) ;
        if (e>=0)
        {
            if (xchange==1) x = x + s1;
            else y = y + s2;
            e = e - ((int16_t)dx<<1);
        }
        if (xchange==1) y = y + s2;
        else x = x + s1;
        e = e + ((int16_t)dy<<1);
    }
}

/*
 * DRAWCURSOR: Draws the cursor at position x, color c: 0=black, 1=white,
 *      2=invert
 */
void drawCursor(uint8_t x, uint8_t c)
{
    uint8_t y;

    for (y = 0; y < 128; y++)
        drawPoint(x, y, c);
}

/*
 * INITIALIZE: Set up the microcontroller registers and initialize global
 *      variables.
 */
void initialize()
{
    uint8_t i;

    // PB7: SCK         in
    // PB6: MISO        out
    // PB5: MOSI        in
    // PB4: VIDEO CS    in
    // PB3: VIDEO REQ   in
    // PB2: VIDEO ACC   out
    // PB1: (unused)
    // PB0: USART XCK   out
    // Note: the USART XCK needs to be configured as an output even though it
    //       is unconnected (required by MSPIM mode)
    PORTB = _BV(PORTB2);
    DDRB = _BV(DDB6) | _BV(DDB2) | _BV(DDB0);

    // PD7: LED         out
    // PD6: (unused)
    // PD5: (unused)
    // PD4: uC DETECT   in
    // PD3: (unused)
    // PD2: SYNC        out
    // PD1: VIDEO       out
    // PD0: USART RX    in
    DDRD = _BV(DDD7) | _BV(DDD2) | _BV(DDD1);

    // SPI: MSB first, slave, mode 00, fosc/4 (5 MHz)
    SPCR = _BV(SPE);

    // TIMER 1: OC1* disconnected, CTC mode, fosc/1 (20MHz), OC1A and OC1B
    //       interrupts enabled
    TCCR1B = _BV(WGM12) | _BV(CS10);
    OCR1A = LINE_TIME;  // time for one NTSC line
    OCR1B = SLEEP_TIME; // time to go to sleep
    TIMSK1 = _BV(OCIE1B) | _BV(OCIE1A);

    // USART in MSPIM mode, transmitter enabled, frequency fosc/4
    UCSR0B = _BV(TXEN0);
    UCSR0C = _BV(UMSEL01) | _BV(UMSEL00);
    UBRR0 = 1;

    // SLEEP: idle sleep mode
    SMCR = 0x01;

    // cursor and trigger off
    x_cursor = 255;
    y_trigger = 255;

    // initialize the sample array
    for (i = 0; i < 200; i++)
        samples[i] = 0;

    // start at first videoline, with regular sync pulses
    video_line = 1;
    sync_on = 0;
    sync_off = _BV(PORTD2);
}

/*
 * VERIFY: Check to make sure the code is running on the correct
 *       microcontroller. If not, flash a warning on the LED forever.
 */
void verify()
{
    uint8_t i;

    // if pin 4 on port D is low, everything's fine!
    if (!(PIND & _BV(PIND4)))
        return;

    // we're executing on the wrong microcontroller!
    DDRD |= _BV(DDD7);
    while (1)
    {
        // flash the LED forever
        PORTD |= _BV(PD7);
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD &= ~(_BV(PD7));
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD |= _BV(PD7);
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD &= ~(_BV(PD7));
        for (i = 0; i < 20; i++)
```

```
            _delay_ms(10);
        PORTD |= _BV(PD7);
        for (i = 0; i < 20; i++)
            _delay_ms(10);
        PORTD &= ~(_BV(PD7));
        for (i = 0; i < 100; i++)
            _delay_ms(10);
    }
}
```