Adafruit Menta ECG Simulator

James P. Lynch

January 25, 2013

Introduction

This document describes in detail the steps required to create an Arduino-based ECG simulator. An ECG simulator replicates the cardiac waveform (Figure 1) that can be measured by attaching three electrodes (RA, LA, RL) to the patient's chest. This ECG signal is only a few millivolts in amplitude.

Figure 1: Typical ECG Signal as Recorded in a Physician's Office

The project was built using an Adafruit Menta kit plus a few additional parts. The Menta kit includes a Arduino ATMega328P microprocessor with 32K of Flash memory and 2K of RAM memory plus an Altoids-type metal case which has enough room to fit a small numeric display, a potentiometer to adjust the heart rate, and three banana receptacles for the patient leads.

The waveform was created by first doing a screen capture of a suitable waveform image from the Internet. This picture file was then digitized using the open source Engauge program from Sourceforge. The resulting text file was further processed by a custom Python program that used linear interpolation to space the samples 1.0 millisecond apart followed by formatting the digitized table into a C Language array construct that could be pasted into the Arduino sketch.

To output an analog waveform on the Arduino Menta, an inexpensive Microchip 12-bit digital-to-analog converter was soldered to the Menta prototyping area. A simple resistive voltage divider was employed to attenuate the D/A signal to the required millivolt levels.

The resulting signal from the Adafruit Menta was then connected to a Texas Instruments ADS1293EVM Evaluation Module which itself demonstrates the operation of their ADS1293 ECG front-end chip (a single integrated circuit that implements all the signal processing normally found in the front end of an ECG heart monitor). The TI software that shipped with the Demonstration Kit was used to display the incoming ECG signal from the ECG simulator which agreed closely with the shape and amplitude of the ECG waveform captured from the Internet document.

While this project was directed solely at generating an ECG signal, the methodology could be used to create just about any waveform you can draw or extract from a document!

The project also shows just how useful the Adafruit Menta kit can be when used as a starting point for a custom, one-off embedded micro-controller design. I needed an ECG simulator and was able to build it for about $60 in parts.

Background

Twenty six years ago, I was in charge of of software development for the world's first color heart monitor, the Mennen Medical Horizon 2000, shown in Figure 2. The CRT display for this instrument was the first to make use of color for alarm conditions, alerts, and so forth.

The Horizon 2000 patient monitor had two Motorola 68000 micro-processor circuit boards, one for signal collection and one for the graphics display and soft-menu system. I wrote the software for the signal collection board and my astute and industrious office mate, Linda, wrote the software for the board supporting the menus and displays. In a design feature rarely seen today, the 68000 boards communicated via a shared, arbitrated dual-port RAM memory.

Figure 2: Horizon 2000 Heart Monitor, Vintage 1986

While developing software for the heart monitor, one indispensable tool I used was a “ECG simulator”. This was a device that created a reasonable facsimile of an ECG signal (that waveform you see on all medical shows). These units were usually battery-powered and the heart monitor's ECG leads were connected to the simulator rather than a patient (or yourself). These ECG simulators can be bought on Ebay for a couple hundred dollars, see Figure 3 for a used ECG simulator that costs $245. An ECG simulator can be very sophisticated, displaying not only the standard “normal sinus rhythm ECG” but the abnormal waveforms as well (arrhythmia, tachycardia, and so forth).

Figure 3: Commercial ECG Simulator

In the years that followed, Mr. Mennen sold his company to a buyer who then moved the entire operation to Israel. I've spent the rest of my career writing software for industrial motor controllers, for a company called Control Techniques.

Single Chip Heart Monitor

As I near retirement, I'm interested in building a heart monitor for myself, you know all DIY and open-source. These days you can get the entire analog front-end for an ECG monitor as an inexpensive integrated circuit from Texas Instruments. The ADS1293 device supports four ECG leads and does all the A/D conversion and signal processing. You can communicate through a simple SPI interface and this should make it straightforward to “isolate” the ADS1293 from the rest of the heart monitor using, for example, Analog Devices transformer-based digital isolators.

To learn all about the TI ADS1293 ECG front-end chip, I purchased a $99 evaluation board from TI, shown in Figure 4.

The ECG leads are wired through the 9-pin D-connector on the far left. A USB cable connects this board to the PC and TI supplies a sophisticated Windows application that lets you modify all the chip's control registers, view the signals, and so forth. The first step in learning about any sophisticated chip is to get a data sheet and an evaluation board.

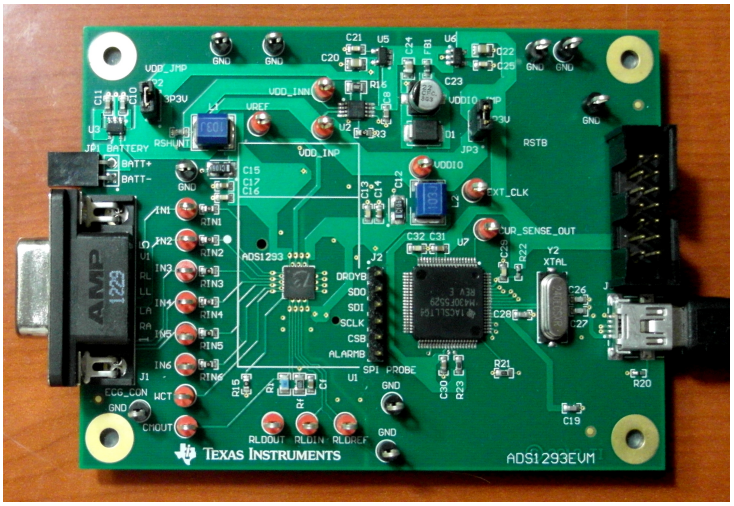


Figure 4: TI ADS1293EVM Demonstration Kit ($99)

Need a ECG Simulator

Now we get to the heart of the matter; to use this TI evaluation board there must be an ECG signal. Why not just hook yourself up with some ECG leads and those conductive pads they use in hospitals? First, Texas Instruments would totally freak out since **there are no protection circuits on this evaluation board.** By this I mean the isolation circuitry, the Zener diodes, the neon-lamps, current-limiting resistors normally used to protect the patient are not present on this board. Second, such a scheme would be cumbersome and inconvenient.

Obviously I could buy a used ECG simulator, but that wouldn't be any fun. How about building one from scratch? I looked at possible solution platforms and a nice starting point is Adafruit's Menta kit, shown in Figure 5. The Menta is a Arduino board with an Atmel ATMega328P 8-bit microprocessor with 32k of Flash memory and 2k of RAM. Notice that Lady Ada designed it to fit into a metal Altoids-style case. Unfortunately the ATMega328P does not have a D/A (digital-to-analog) converter which will be needed to generate the analog ECG waveform. The Menta with the case is $35.00, delivered as a kit.

Figure 5: Adafruit Menta Arduino Kit

Looking at the Menta and applying a little arm chair engineering, we'll need to fit three banana receptacles on the top to attach the three ECG leads, a pot with a knob to adjust the heart rate, and a small 4-digit 7-segment numeric display to show the dialed-in heart rate.

The Menta has a breadboard area which can be used to fit a small D/A converter chip. A resistor voltage divider network will attenuate the 5 volt ECG signal from the D/A converter down to a couple of millivolts, which is the amplitude of a human ECG signal detected via chest electrodes.

The Menta can be powered via a cheap 9-volt Wall Wart using the power connector on the left. Programming is accomplished using an inexpensive FTDI Friend board ($14.75 from Adafruit) using your computer's USB port. Becky Stern, an Adafruit employee and New York City artist, has a very nice introductory video about the Menta here:

http://www.youtube.com/watch?v=opuD2h4puSk&feature=player_embedded

A block diagram showing the basic components of the Menta ECG Simulator is shown in Figure 6.

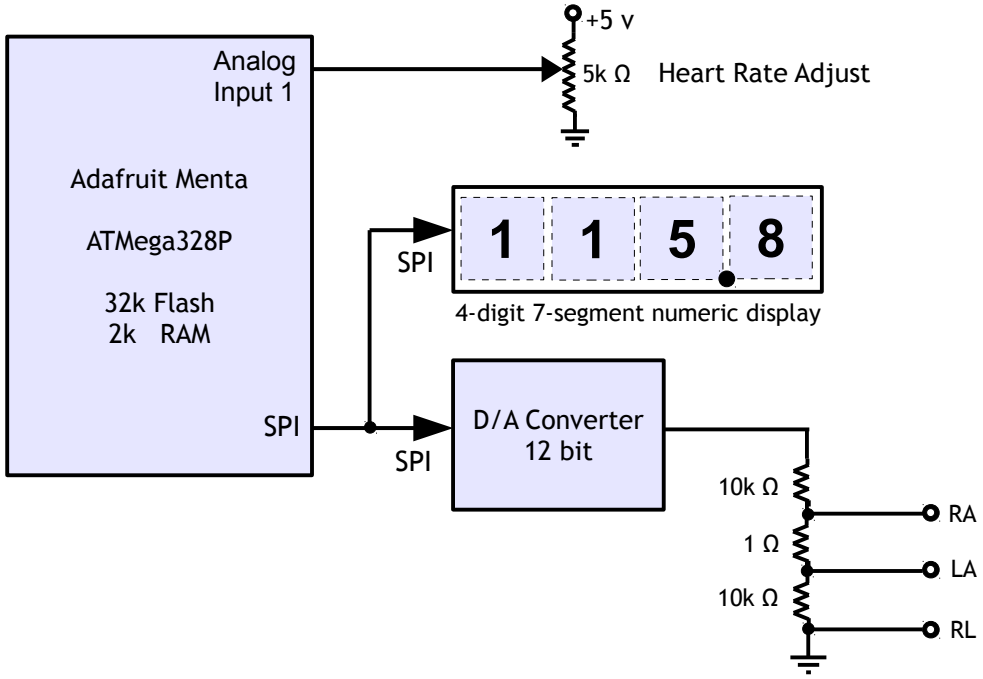


Figure 6: Menta ECG Simulator Block Diagram

Acquiring the Parts

The $35.00 Adafruit Menta is a kit that you must assemble. In the spirit of and even better than Heathkit, Adafruit has a detailed tutorial on their web site that shows how to assemble the Menta. In my case, it took less than an hour to solder the kit together. I actually bought two kits, one for the ECG Simulator and one for bread boarding (where I fitted the headers for the I/O ports).

For the D/A converter, I selected the Microchip MCP4921 single channel D/A chip, which communicates via the SPI interface. The resolution of the MCP4921 is 12-bits; it takes two sequential SPI 8-bit transmissions to send the 12 bits plus four configuration bits.

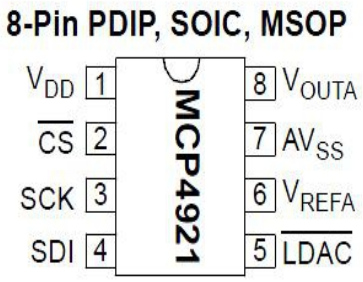


Figure 7: Microchip MCP4921 D/A Converter

The D/A converter will operate at a rate of 1000 updates per second, or in other words 1.0 millisecond per sample. The D/A signal will be updated as part of an on-board Timer2 interrupt handler. The execution time to update the D/A via the SPI interface is only 63 microseconds (author's measurement). Not long after I acquired the Microchip D/A chip, Adafruit offered a dual-channel D/A device on a breakout board for $4.95. I'm sure that this device would be also perfect for this application. The Microchip MCP4921 can be ordered from Digikey for $2.36. The pin layout for the MCP4921 is shown in Figure 7 above.

For the 4 digit heart rate display, I considered three possible solutions. In Figure 8, the top device is the Adafruit 0.56" 4-Digit 7-Segment Display w/I2C Backpack for $9.95. This display's on-board controller makes it easy to communicate via SPI and the numbers are large and bright. However, you can see from Figure 8 that it's a bit too big for the Menta cover.

The middle device is the Adafruit Monochrome 1.3" 128x64 OLED graphic display for $24.50. It's small and thin and the interface is SPI so it looked like a good choice. It proved unsuitable because the interface is write-only so you can't read back the internal graphic RAM. This means that Adafruit's software driver had to maintain a complete copy of the display's graphic RAM on the ATMega328P chip. Worse yet, the driver Adafruit prepared writes the entire graphic RAM for any command, even if you only wanted to change one pixel. I measured the execution time to update the entire graphic RAM and it was longer than the expected D/A sample period of 1.00 msec. Reluctantly I set this one aside for future projects.

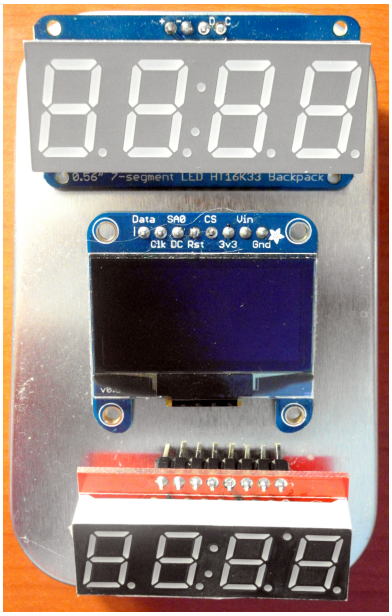


Figure 8: Possible Display Devices

The bottom display device is the Sparkfun COM-09764 7-Segment Serial Display – Blue for $12.95. It also has a controller that makes it work with a simple SPI interface. Note that the size looks appropriate for the Menta cover, so it's mountains over Manhattan for the numeric display!

For the pot to adjust the heart rate, I found the typical pot available from Radio Shack (and Adafruit) to be too thick for the Menta case. Searching the Jameco catalog, I found a pot intended for circuit board mounting that is somewhat thinner. The Jameco pot is the bottom one in Figure 9. This is the Panel Control - 22MM-ST-CP 3 (part number: 1998141) for $3.39. The metal clip can be popped off to give a much thinner profile (this is a 5k pot).

I also purchased a knob from Jameco, this is Knob ¼" Shaft, Metal, Round, Silver for $0.99 (part number: 162481).

The banana receptacles, resistors, and capacitors are stock items at Radio Shack.

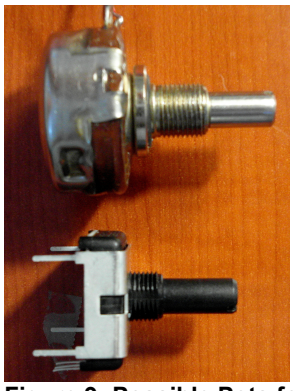


Figure 9: Possible Pots for HR Adjust

Here is a **Bill Of Materials** for the parts required for the project:

Quantity	Vendor	Part Number	Price (each)	Notes
1	Adafruit	Menta	$35.00	Arduino with Case
1	Microchip	MCP4921-E/P-ND	$2.36	D/A Converter
1	Sparkfun	COM-09765	$12.95	7-Segment Serial Display
1	Jameco	1998141	$3.39	5K Pot - Panel Control 22MM-ST-CP
1	Jameco	162481	$0.99	Knob, 1/4" Shaft, Metal,Round
1	Jameco	25523	$0.08	0.1 uF bypass capacitor, 50V, 20%
2	Radio Shack	271-1335	$0.24	10K resistor, ¼ watt
1	Radio Shack	271-1301	$0.24	10 ohm resistor, ¼ watt
3	Radio Shack	274-725	$1.60	Banana Jack

ECG Simulator Schematic

Figure 10 is the final schematic for the Menta ECG Simulator project. LadyAda's Eagle schematic for the Menta was used as a starting point and I simply added the parts required to complete the project. I did not attempt to make a custom circuit board from this schematic since the intent is to simply solder the D/A converter, etc. directly to the breadboard area of the Menta.

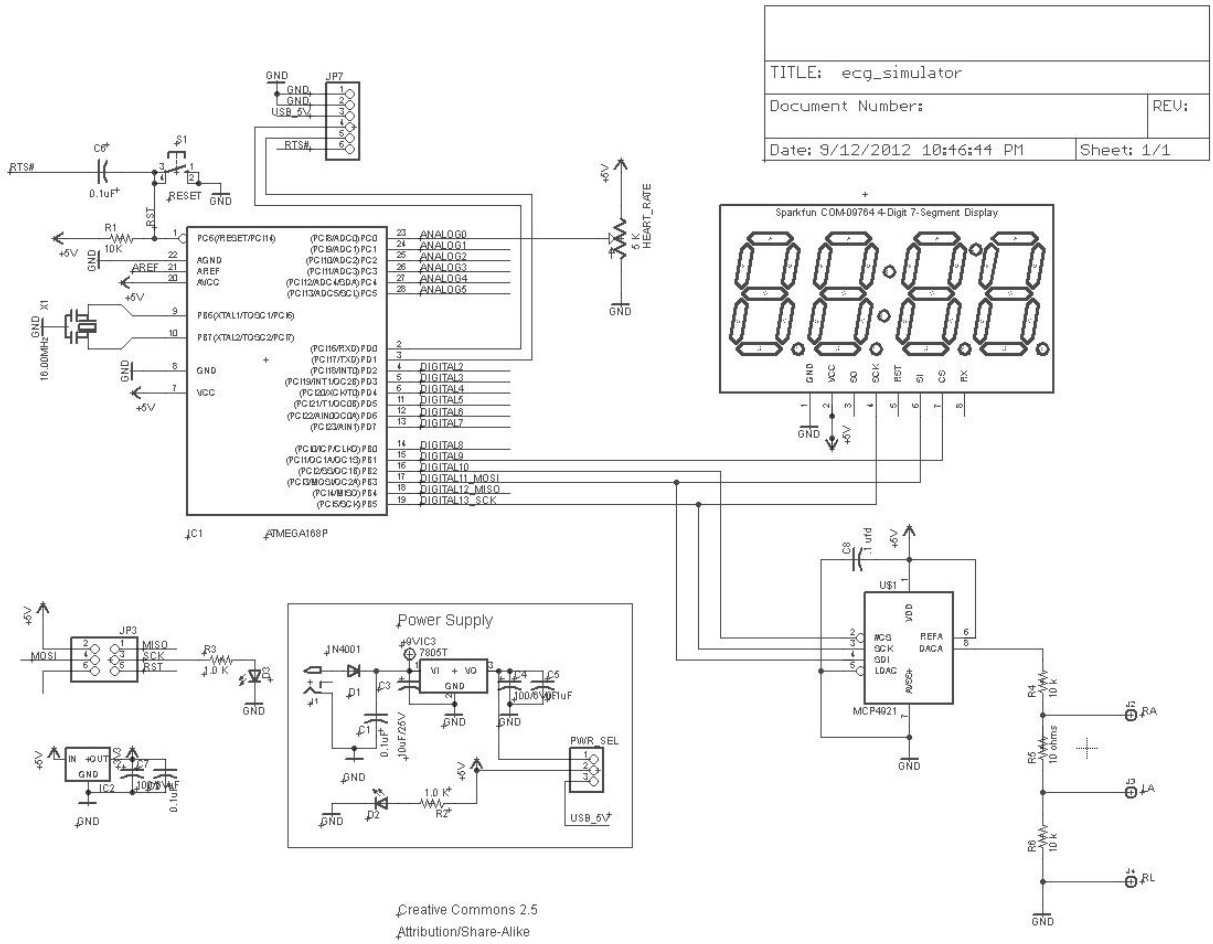


Figure 10: Menta ECG Simulator

Menta ECG Simulator Wiring Diagram

Figure 11 shows the wiring of the Menta ECG Simulator. All wiring is routed on the top (component) side of the Menta circuit board and most soldering is on the bottom side of the board. Solder joints are identified as small black circles. I elected to simply solder the MCP-4921 D/A converter chip directly to the board, but one could easily substitute an 8-pin IC socket instead.

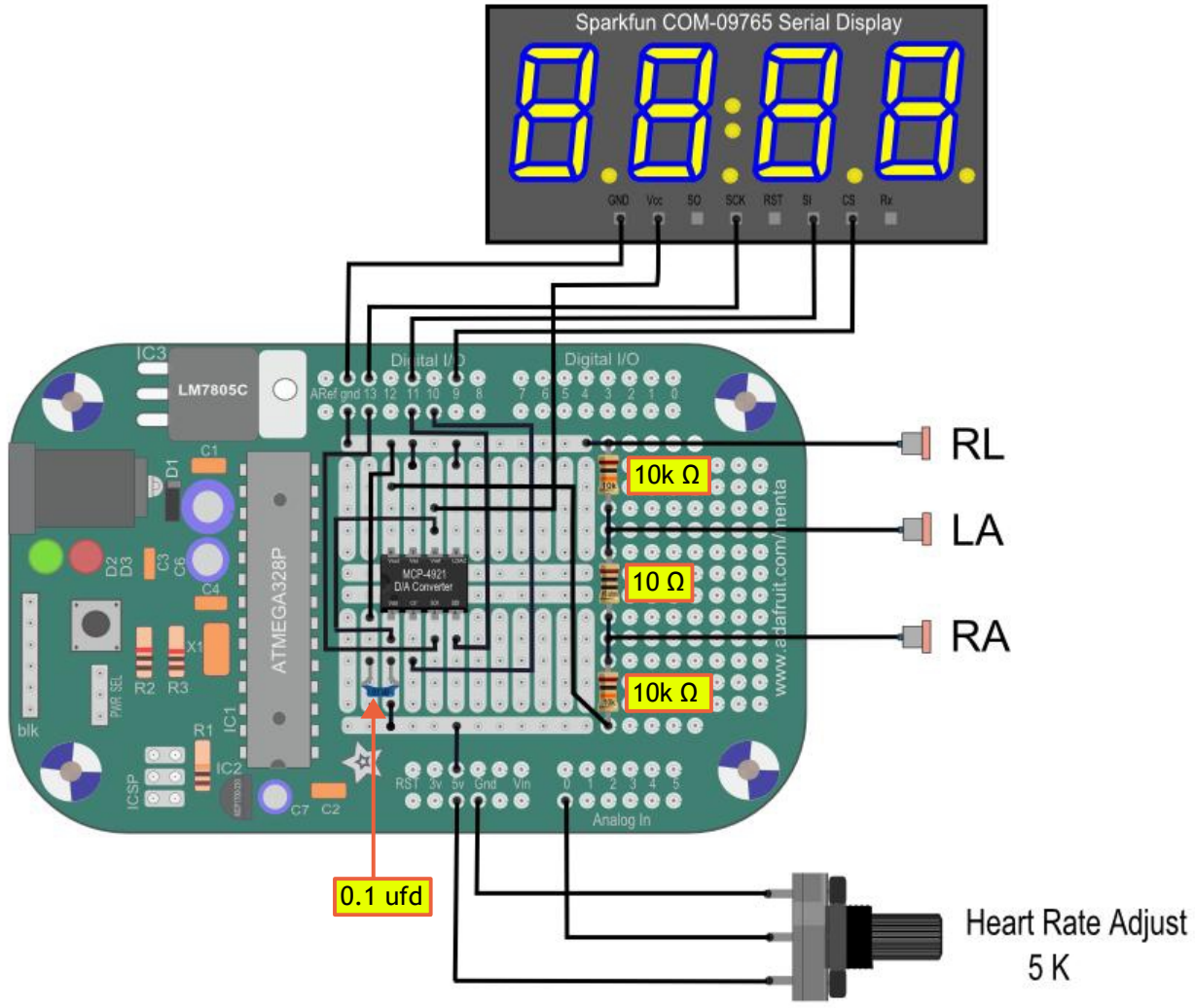


Figure 11: Menta ECG Wiring Diagram

I utilized a “Solderless Breadboard Jumper Wire Kit” to wire the prototype. This is a plastic box with a bunch of 22AWG solid hookup wire, stripped on both ends and in various lengths. I ended up cutting many of these to length and stripping one end, but it's still a time saver. Maker Shed and Sparkfun carry these hookup wire kits for $6.25.

Wiring the Menta ECG Simulator Board

Figure 12 shows the completed ECG Simulator prior to installation into the Altoids case.

I added a couple of stake pins for ground and the output of the D/A converter so operation could be verified with an oscilloscope. Since the banana jacks for the ECG leads have to be bolted onto the case, they can't be soldered to the green outputs from the voltage divider at this point.

The Adafruit Menta is specifically designed for the type of build you see here. The I/O ports and power/ground access points are all doubled up, all holes are plated-through, and all pads are tinned for easy soldering. If you can build a circuit with a breadboard, you can build a one-off prototype with the Adafruit Menta!

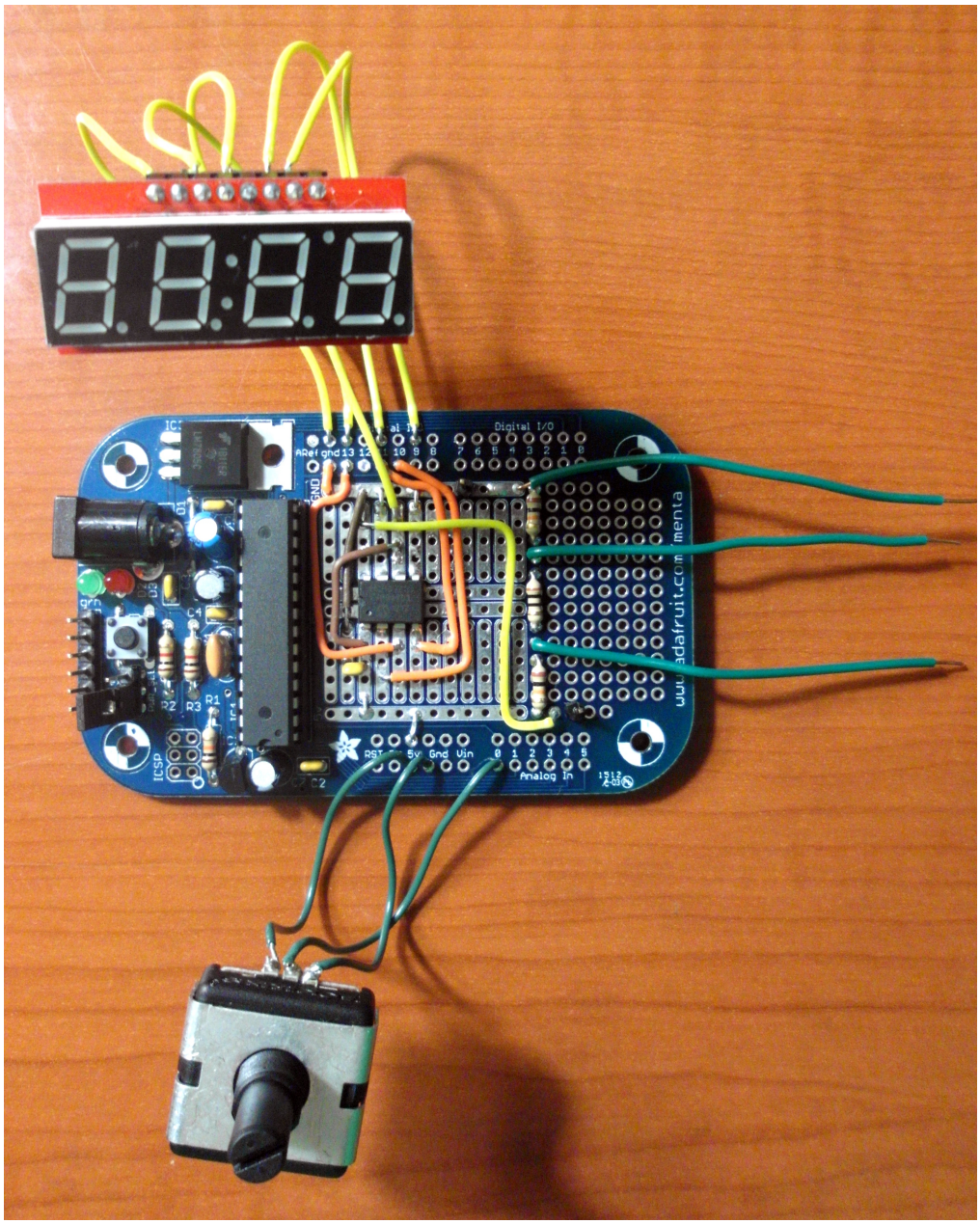


Figure 12: Fully Assembled Menta ECG Simulator

Drilling the Menta Mounting Holes

Rather than trying to measure and locate the circuit board mounting holes on the bottom of the Altoids metal case, it's a lot easier to drop the Menta board into the case and use a center punch to mark the holes, as shown in Figure 13. A center punch uses a spring-loaded pin to dimple the metal surface and thus provides a good starting point for a drill bit.

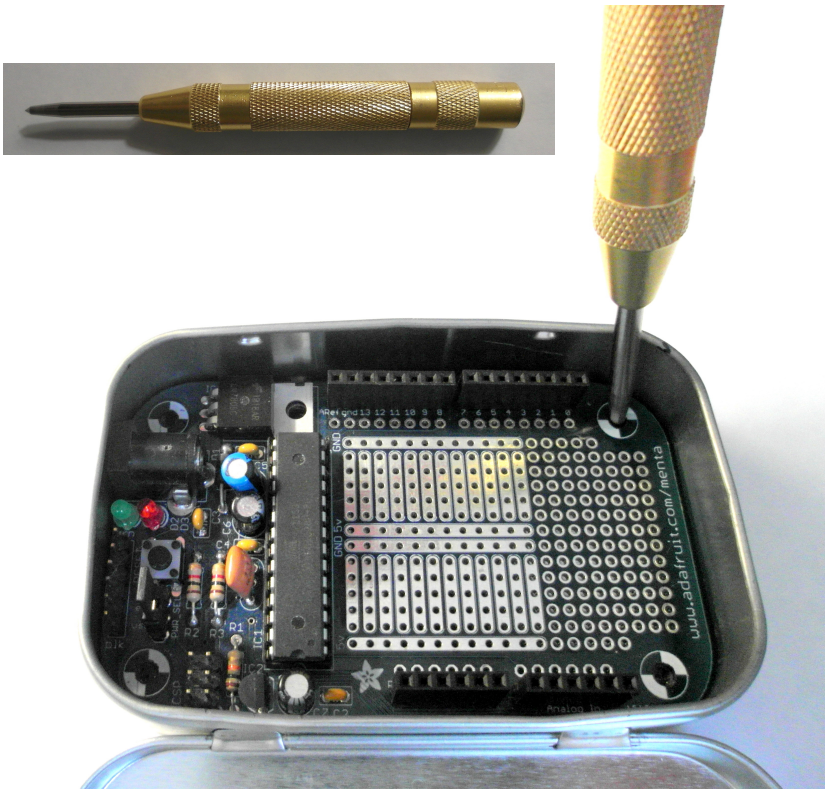


Figure 13: Using a Center Punch to Locate Drill Holes

Figure 14 shows the dimples created by the center punch's spring-loaded pin driver, ready for the drill press.

Figure 14: Starting Dimples for Drill Press

Now the circuit board holes can be drilled with a desk-top drill press. The Adafruit Menta holes measured 7/64” diameter so a 1/8” drill bit seemed suitable. Figure 15 shows the drill press in action. The resulting holes on the bottom side of the Altoids tin have ragged “burrs” after drilling so these can be ground down using a Dremel MotoTool with a flat grinding wheel (Figure 16).

Figure 15: Use 1/8" drill bit for the circuit board mounting holes.

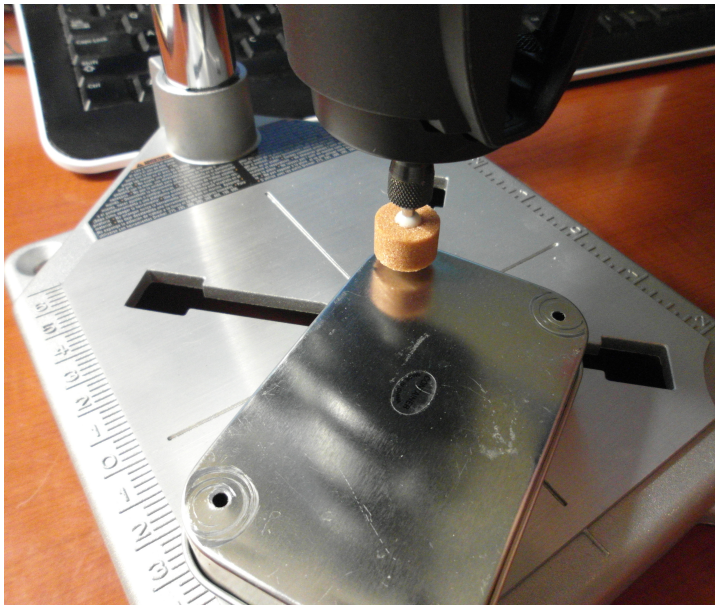


Figure 16: Use Dremel Tool Grinding Wheel to remove burrs

Figure 17: Four completed 1/8" mounting holes for the Menta circuit board.

Design the Front Panel

One very nice tip from the Adafruit blog (http://www.adafruit.com/blog/) was this excellent tutorial by Phil at Jumper One concerning making nice front panels at home.

http://jumperone.com/2013/01/how-to-make-diy-front-panel/

Based on Phil's suggestions, the first thing to do is design a front panel layout that locates all the holes, etc. A good drawing tool to use is **Inkscape**, an open source and free vector-based drawing system. Inkscape can be downloaded from here: http://www.inkscape.org/download/

I purchased "*The Book of Inkscape*" by Dmitry Kirsanof (one of the current developers of Inkscape) and spent a couple of days learning just enough to create some pretty fancy drawings.

Figure 18 illustrates an Inkscape-created stick-on label for the front panel.

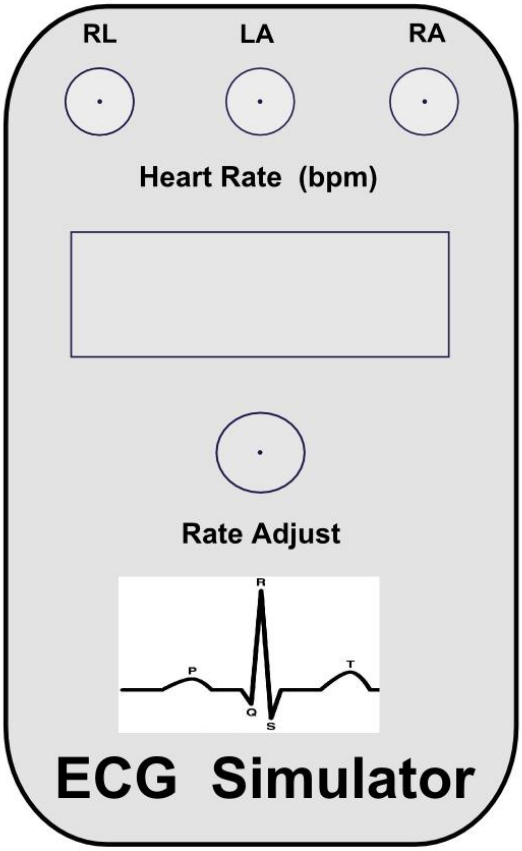


Figure 18: Front Panel Artwork

Figure 19 below shows a dimensional drilling diagram of the front panel (the top cover of the Altoids tin) which locates the holes for the banana jacks, potentiometer, and the 7-segment display cut-out. The design takes into account the underneath sizes of the banana jacks, display, and potentiometer so that these parts don't interfere with each other (the underneath outlines are shown as dashed lines).

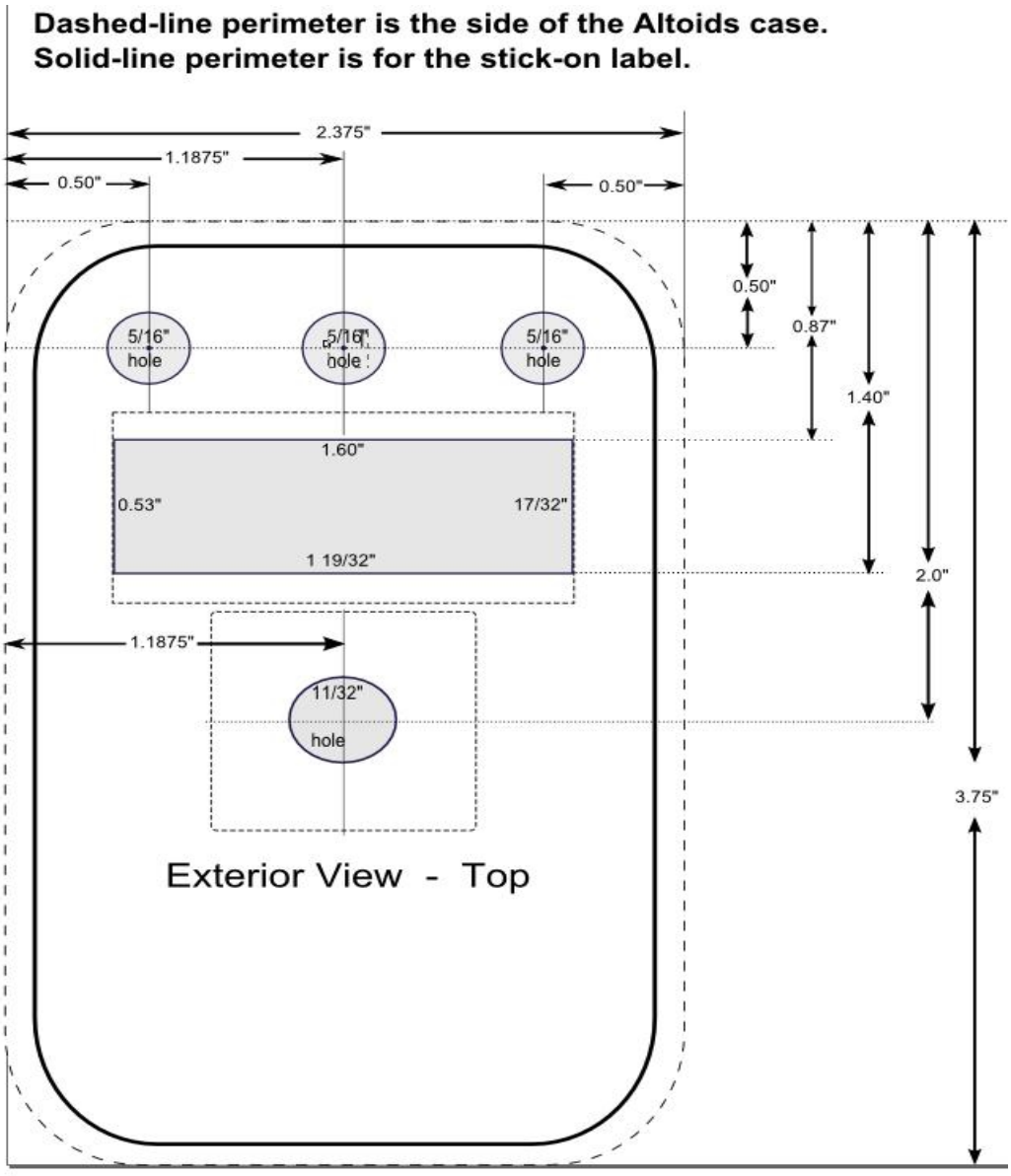


Figure 19: Drilling Diagram for the Front Panel

Drill the Front Panel

Print out the front panel label on ordinary printing paper. Cut out the outline with scissors, then use a hobbyist razor knife and a straight edge to cut out the rectangular hole for the 7-segment display.

Using masking tape, carefully affix the drilling template to the Altoids case front cover as shown in Figure 20. Using a “magic marker and a straight edge, outline the 7-segment display rectangle as shown in Figure 20. As illustrated in Figure 21, center punch the three banana jack holes, the rate potentiometer hole, and a hole in the center of the 7-segment display rectangle (for an opening to get the nibbling tool started). This will be used to “nibble” the rectangular outline for the display. Figure 22 shows the Altoids front cover ready for drilling and nibbling.

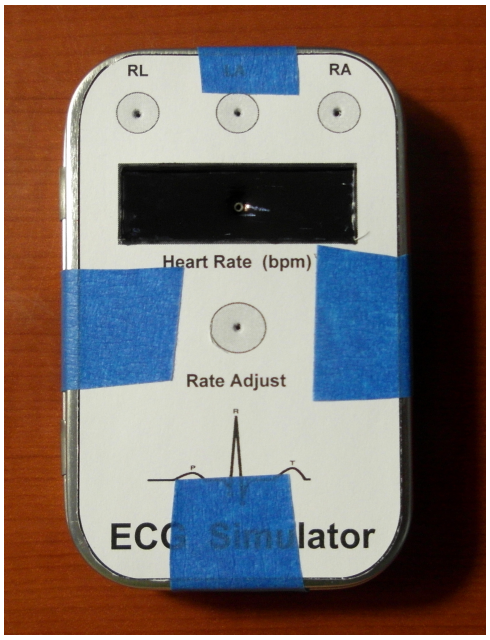


Figure 20: Affix the drilling template

Figure 21: Center punch all holes

Figure 22: Front Cover ready for Drilling

While these holes can be drilled with a hand-held electric drill, a desk-top drill press is more suitable for this task. Figure 23 shows a banana jack hole being drilled. A large hole (11/32”) should be drilled in the center of the 7-segment display rectangle to serve as an entry point for the Radio Shack nibbling tool (Figure 24).

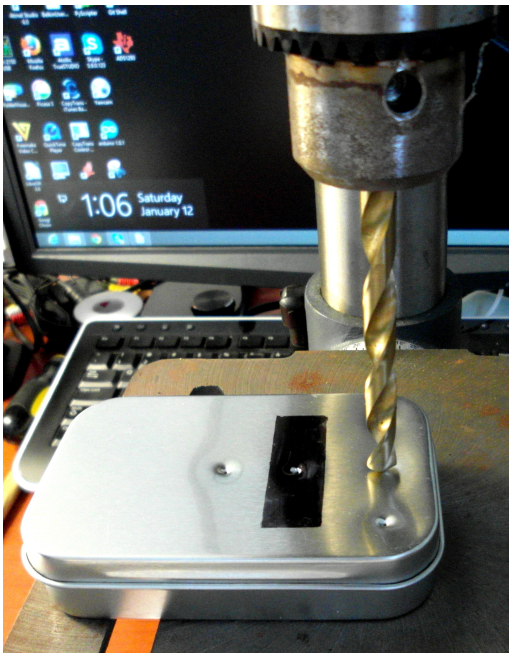


Figure 23: Drilling the banana jack holes

Figure 24: Radio Shack nibbling tool

Step drills, as shown in Figure 25, are extremely useful in widening holes and removing burrs. These can be acquired from Amazon or eBay. Normally they don't need a pilot hole if a spring-loaded center punch is used to locate the hole. The ones the author procured have “flats” on the shaft that eliminate drill chuck slippage.

Figure 25: Step drills are very handy!

The Radio Shack nibbler tool, shown earlier in Figure 24 above, chops a tiny 7/32” x 1/16” rectangular bite out of the metal it is cutting. Basically you insert the cutting tip down into the pilot hole and start nibbling away as shown in Figure 27. If you are careful, a fairly decent cut can be achieved. I cleaned up the final cut with a flat file and then ensured that the 7-segment display will fit tightly through the opening. The finished product is shown in Figure 26 below.

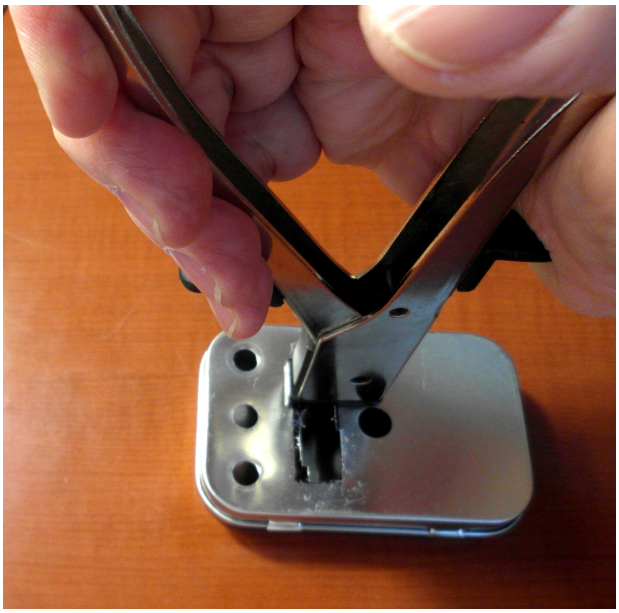


Figure 27: Nibbling the 7-segment display rectangular opening

Figure 26: Altoids case with front panel drilled and nibbled

Cutting the Power Access Port

The last bit of drilling is to provide the access for the power connector. Figure 28 shows the drilling diagram; a 5/16” hole was drilled and the Radio Shack nibbler was used to form a 1/2” rectangular cutout. If the banana jacks are defined as the top, then this power access port would be located bottom left.

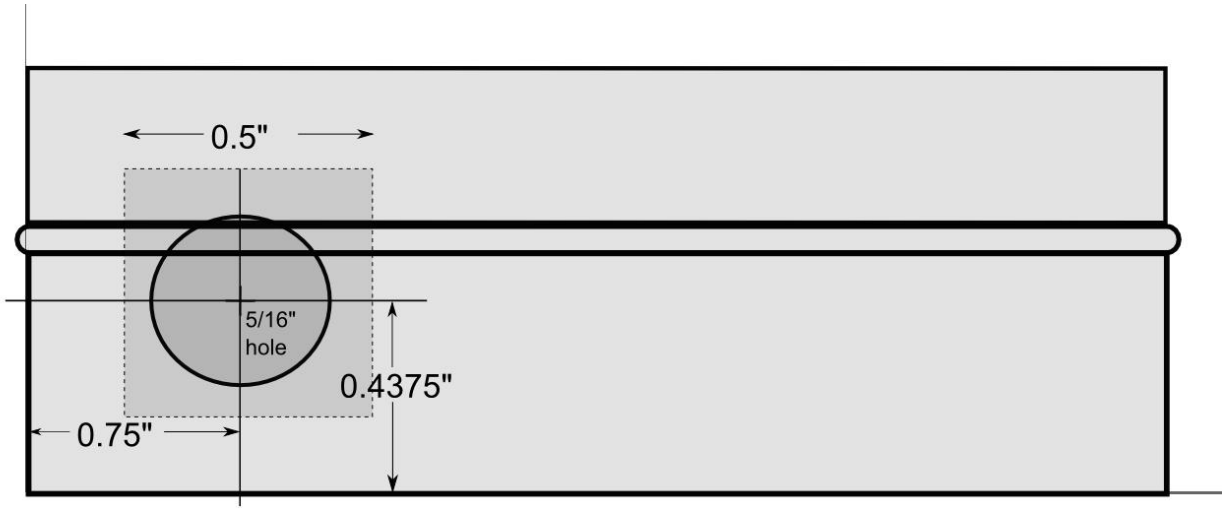


Figure 28: Drilling diagram for access to power jack

Figures 29 and 30 show two different drill bits being used.

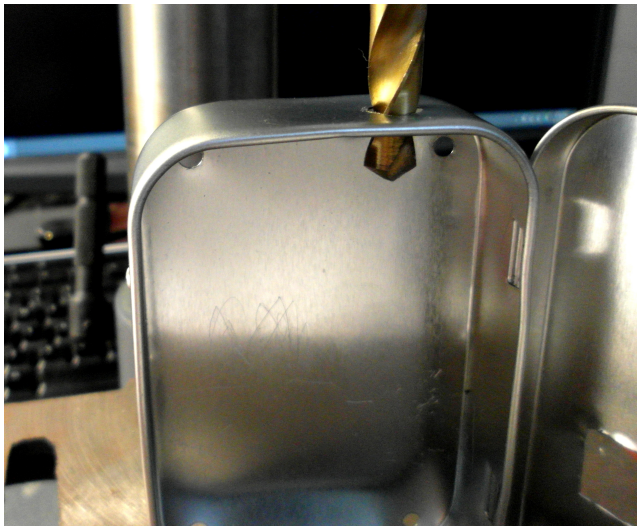


Figure 29: Start with a drill bit

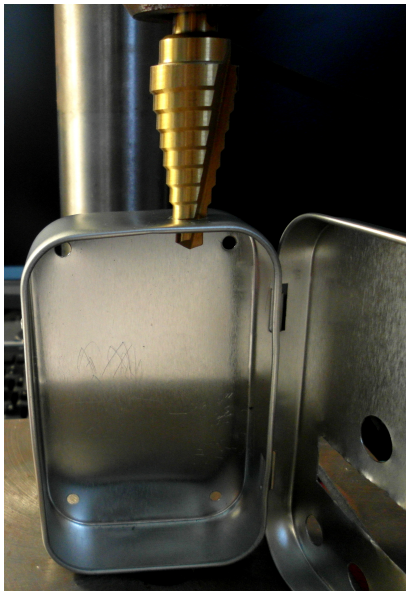


Figure 30: Widen with a step drill bit

After doing some nibbling, the square hole was cleaned up with a hand file (Figure 31). The finished product is shown in Figure 32.

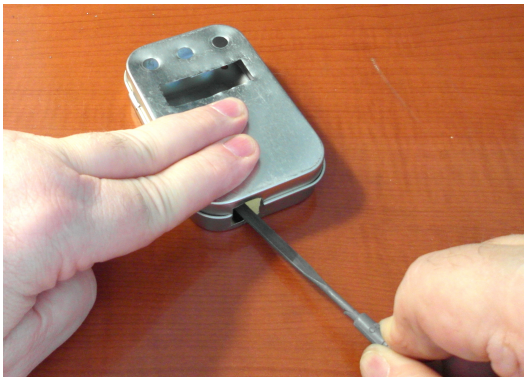


Figure 31: Clean hole with a hand file

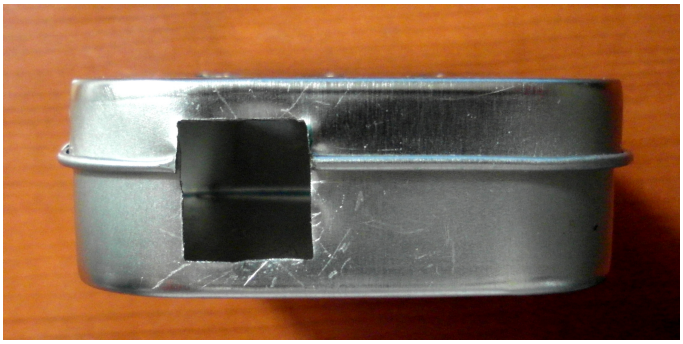


Figure 32: Completed access hole for the power connector

You might think that this access port is too large, but it enables one to see the two LEDs on the Menta board (power and communications).

Create and Attach the Front Panel

While I had considered using Press-Type lettering and a clear lacquer top coat to mark the top of the Altoids tin, a much better solution is to print the label on white label stock and then cover that with single-sided laminating sheet. These items are available from Amazon as shown below.

"Avery® White Full-Sheet Labels for Inkjet Printers with TrueBlock(TM) Technology, 8-1/2 inches x 11 inches, Pack of 25 (8165)"
Office Product; $9.98

"Scotch® Laminating Sheets LS854SS-10, 9 Inches x 12 Inches, Letter Size, Single Sided"
Office Product; $7.97

On the Avery 8165 label stock, print out the front panel label as shown on the left in Figure 33. Using sharp scissors, carefully cut out the label outline. The holes for the banana jacks, potentiometer, and display rectangle can be cut freehand using a sharp hobbyist knife and a straightedge.

Likewise, the label can serve as a template to mark and cut out the clear plastic laminate sheet using the Scotch LS854SS-10 laminating sheets, as shown on the right in Figure 33.

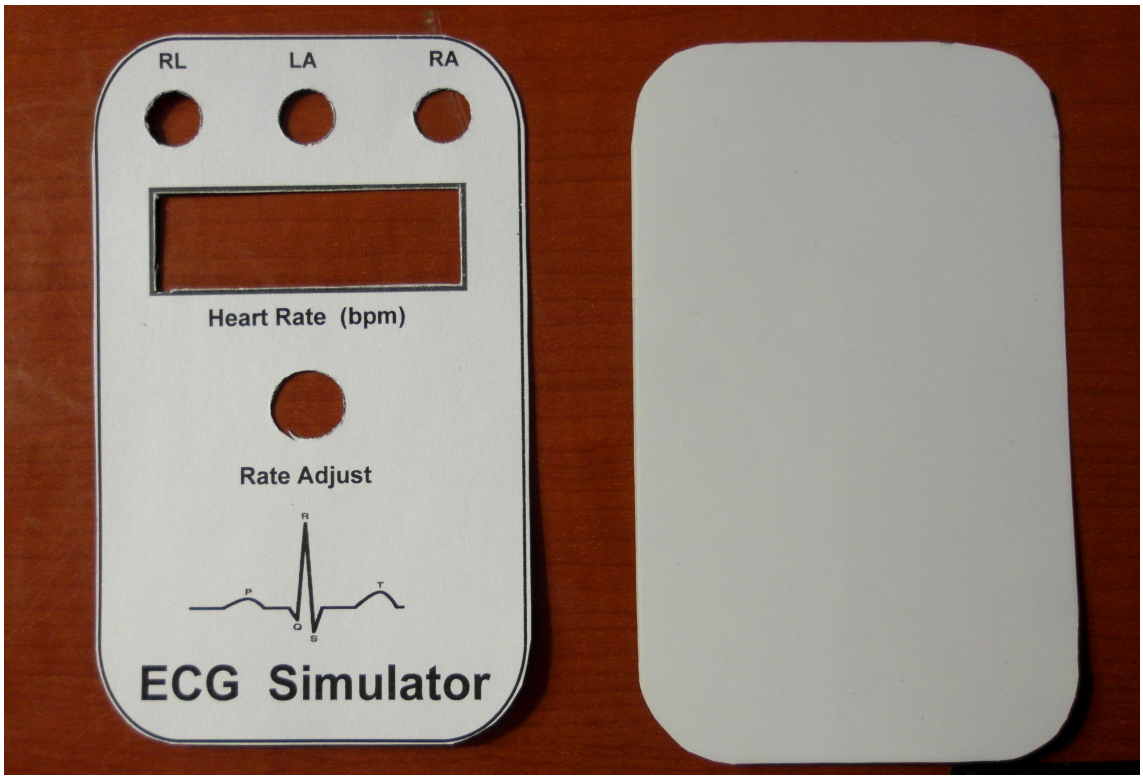


Figure 33: Printed and cut label (left) and laminating sheet (right)

Remove the paper backing from the laminating sheet (right in Figure 33) and apply it to the top of the label (left in Figure 33). Using a sharp hobbyist knife, cut out the holes and the rectangular cut-out for the display again. Now we have a stick-on label with a laminated plastic sheet (Figure 34) that will protect the lettering for a long time (and look very nice too).

To actually stick the label on the case, I temporarily installed one banana jack and potentiometer using masking tape to help register the label (you only get one try at this). Remove the bottom paper backing from the label and stick it on the Altoids tin.

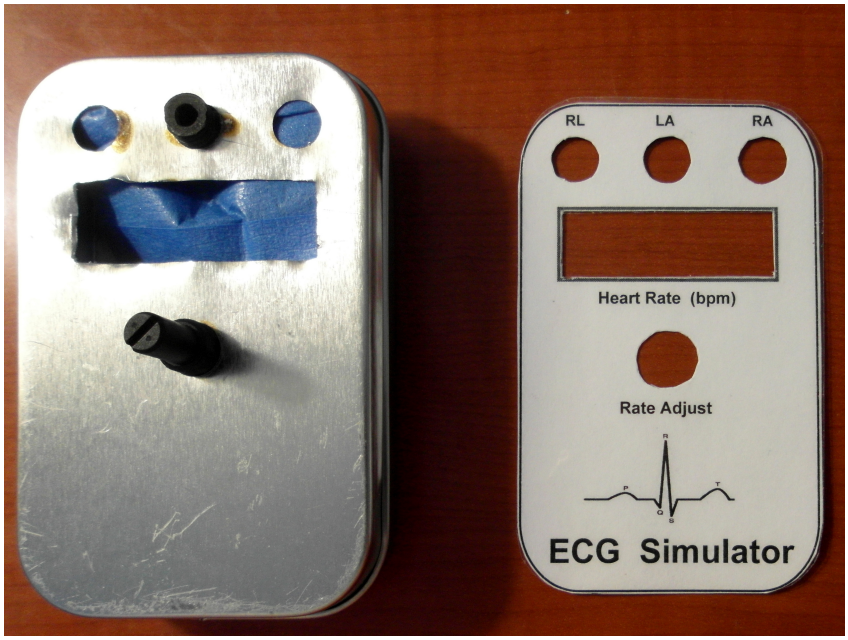


Figure 34: Altoids tin ready for label application.

Figure 35 below shows the Altoids tin with the label attached. Any little mismatches or anomalies around the holes will be covered by the hardware when the banana jacks and potentiometer are installed.

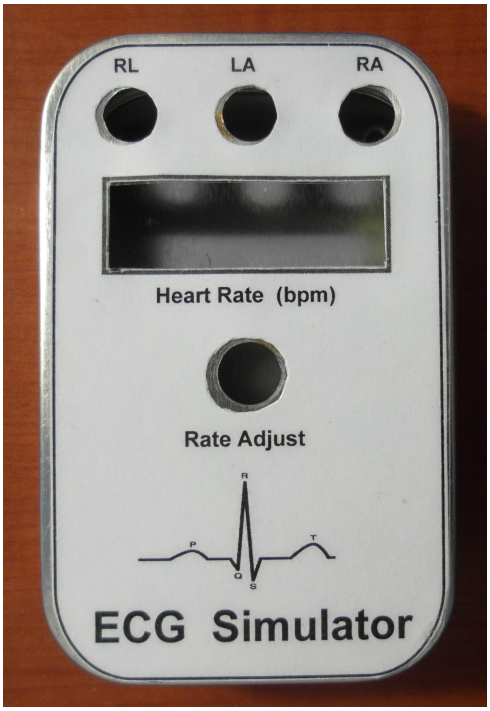


Figure 35: Front panel label installed

Insulating the Altoids Tin Bottom

The inside bottom of the Altoids tin has to be insulated prior to mounting the Menta circuit board. While the Scotch single-sided laminating sheets would be satisfactory, I elected to use heavier plastic stock that is used as front and back covers for report binding. Using the label as a template, the resulting plastic label can be marked for the four mounting holes and then these holes can be cut out using a razor knife. This plastic insulating insert is shown on the right in Figure 36 below. Just remember to install this insulating sheet before dropping the Menta circuit board into the case.

Figure 36: Insulating plastic sheet covers bottom of Altoids tin.

Installing the 7-Segment Display

While the Sparkfun display is a very nice product, it appears that it was designed for bread boarding only. There is no obvious way to mount it in the Altoids case. I elected to epoxy “wings” to the sides of the display and then epoxy this assembly to the case (from underneath).

I used some 1.8” x 3.2mm ABS square rod cut to 1 1/2” for the wings. This stock can be procured from http://www.hobbylinc.com/

PLS90351	Square Rod ABS 1/8 (5)	$4.59

Before attempting to epoxy this rectangular bar stock to the display, it's wise to roughen two adjacent edges with a file before gluing. This is a two-step process; first two “wings” are epoxied to the sides of the display (as shown in Figure 37), and then this assembly is epoxied to the to the Altoids case from underneath.

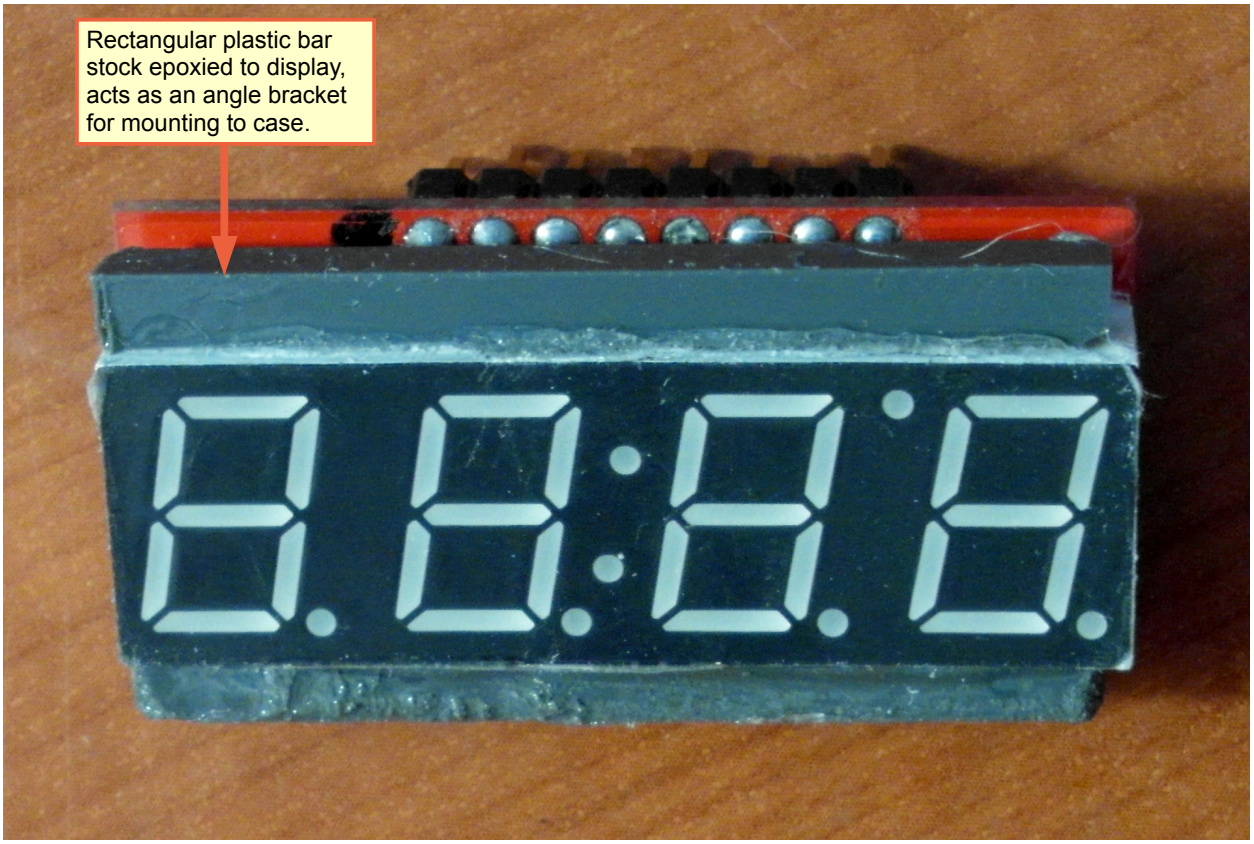


Figure 37: Two rectangular plastic wings epoxied to the display.

In Figure 38 below, you can see how the “wings” are epoxied to the Altoids case. With commonly available 5-minute epoxy, several hours is required for the glue joint to harden

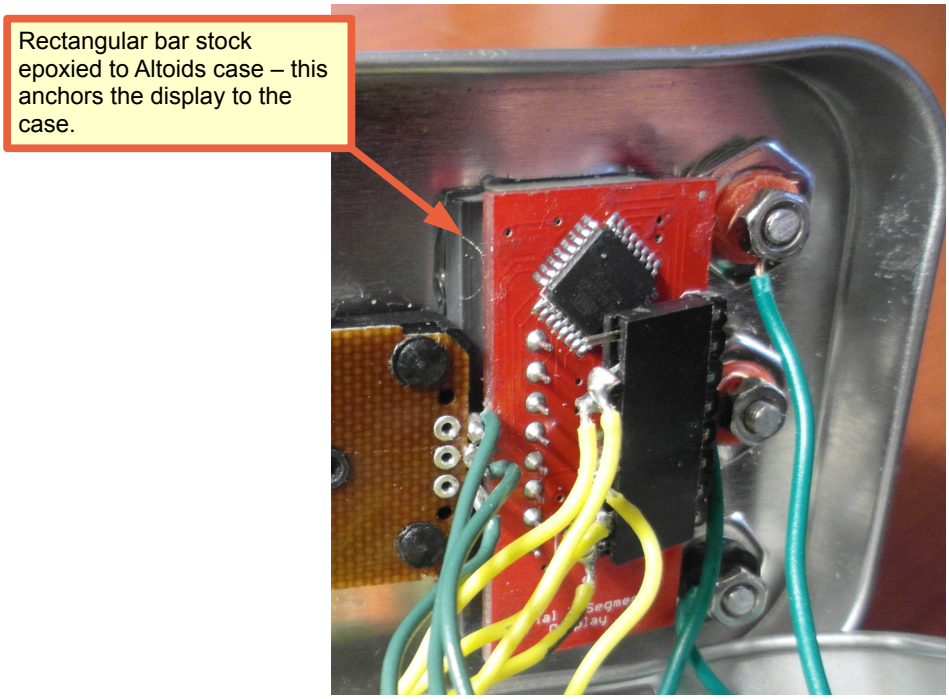


Figure 38: Display is epoxied to the Altoids tin

The Banana Jacks Need Some Surgery

The stock Radio Shack banana jack is too long and would contact the Menta circuit board if installed unmodified. Basically it has to be shortened.

Disassemble the stock banana jack and grip the bottom part (the plastic threaded part that will bolt to the case) with vice grip pliers. Be sure to leave the washer and threaded nut on the bottom assembly because they will be used to “restore” the plastic threads after cutting through. Just backing the threaded nut off after cutting cleans up the threads damaged by the cutting tool.

I used a Dremel MotoTool with a cut-off wheel to cut through the plastic threads. Try to leave two or three threads on the bottom part. Figure 39 shows the bottom part of the banana jack being shortened. As you can see, the nut is still installed so you can back it off after cutting and thereby restore the threads.

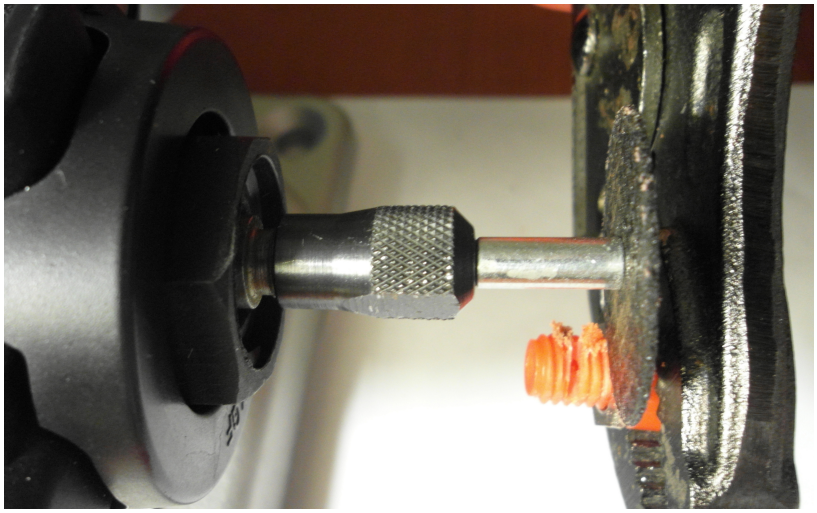


Figure 39: Shortening the banana jack bottom part with a cut-off wheel (vice grip pliers used to hold the banana jack)

Figure 40 shows the result of shortening the banana jacks. The stock Radio Shack part is shown for reference.

Figure 40: Banana jack bottom part shortened

The banana jack metal post has to be shortened too. Leaving one nut threaded as shown in Figure 41, use a Dremel MotoTool cut-off wheel to cut the post, leaving three or four threads.

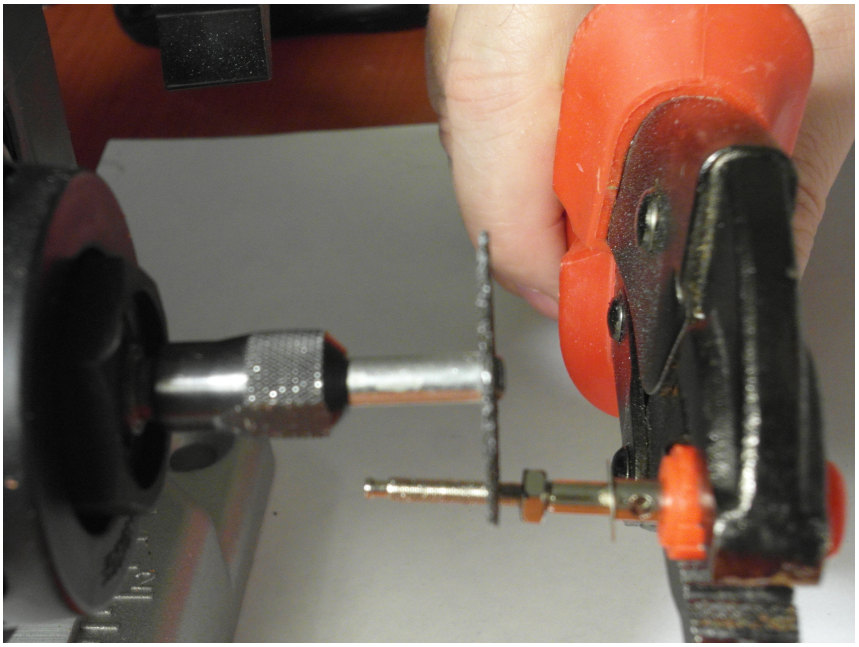


Figure 41: Shorten the banana jack post also.

Figure 42 shows the modified banana jacks (with the original for comparison). There's just enough threads in the post to back the nut away enough to loop and attach the ECG wires from the circuit board.

Figure 42: Modified Banana Jacks (quite a bit shorter)

Install the Menta Circuit Board

For the circuit board mounting hardware, I used M3 x 8mm machine screws, M3 flat washers, and a M3 hex nut. These fit the mounting holes nicely.

To insulate the bottom of the circuit board from the metal case, I cut a rectangular piece of thin plastic using the Menta board as a template. Plastic report covers work well for this purpose. An Exacto knife can be used to make cut-outs for the machine screws if a paper hole punch is not available.

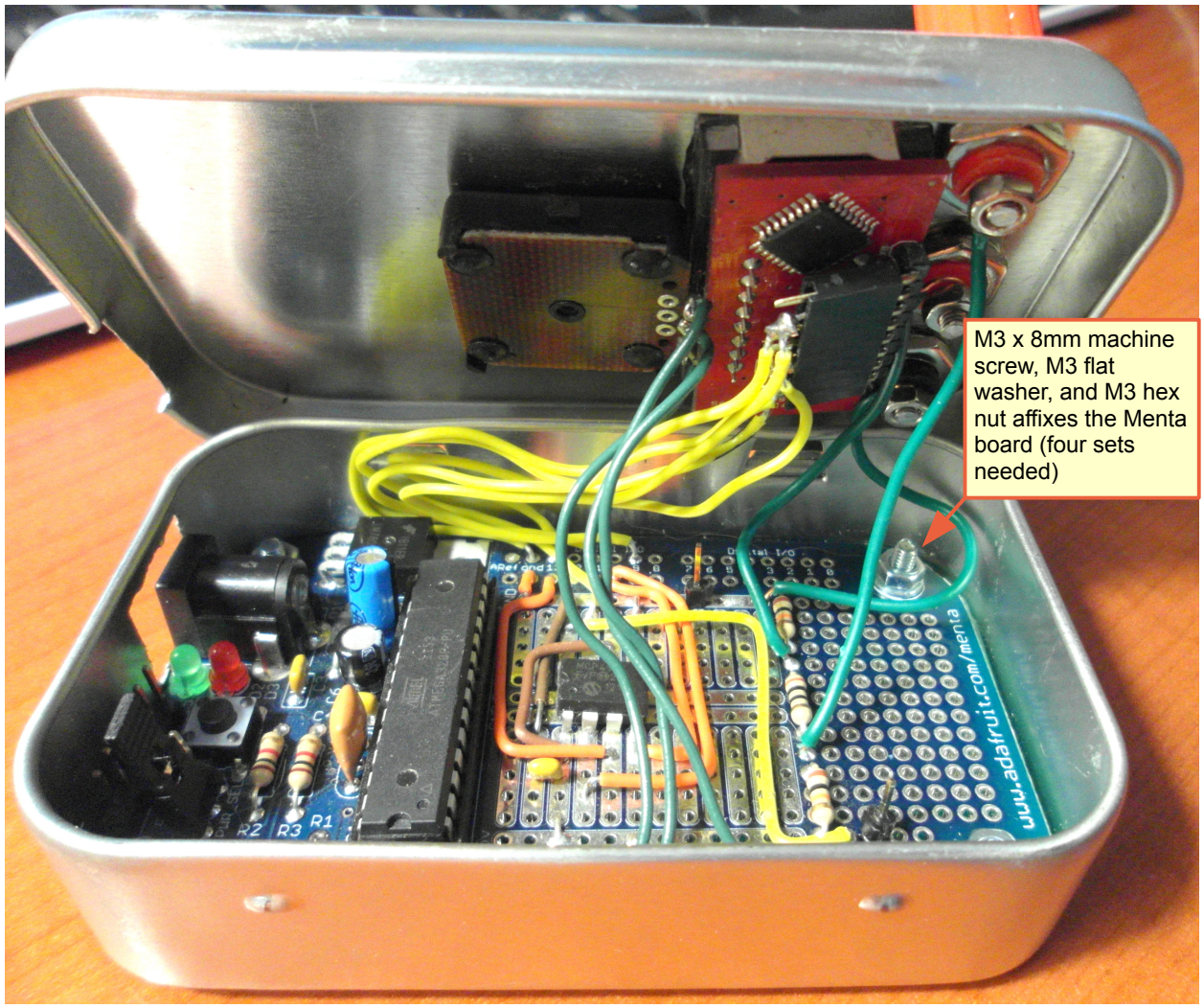


Figure 43: Menta board mounted in Altoids case

Please, please, please don't forget the plastic insulating sheet placed below the circuit board. If you forget this, the Menta board will short circuit!

Install the Banana Jacks and the Potentiometer

The last step in assembly is to mechanically mount the banana jacks and the potentiometer, as shown in Figure 44. I simply formed a small loop at the end of the ECG wires and anchored this over the threaded post with the nuts supplied with the banana jack.

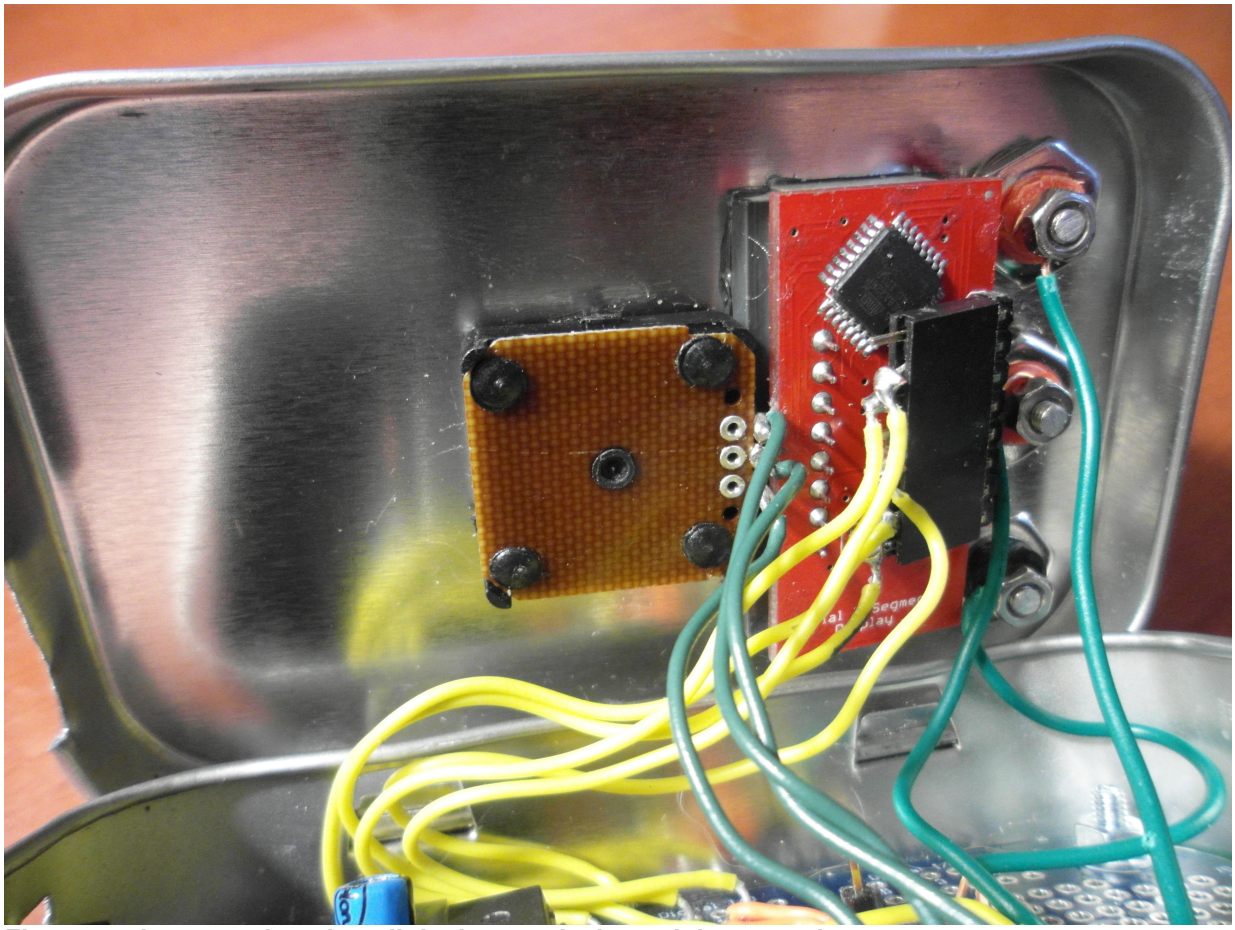


Figure 44: Last step is to install the banana jacks and the potentiometer

You might notice that I used a plug to connect to the Sparkfun 7-segment display. This is a Jameco product:

Jameco	70755	.100” (2.54 mm) female straight header receptacle	8-contacts	$0.55

To provide adequate clearance, I bent the header pins soldered to the Sparkfun board 90 degrees.

The potentiometer is fastened with a nut and a knob is attached and anchored with a tiny allen wrench.

The Completed Hardware

Before closing the Altoids case, make sure that the Adafruit Menta circuit board's PWR_SEL jumper is set for external power (not USB driven). The power jumper is to the left of the Reset button, as shown in Figure 45. To select the external power via the jack, set the jumper to the right as shown.

Figure 45: Set PWR_SEL jumper to External Power (closest to the Reset button)

Figure 46 shows the completed Adafruit Menta ECG Simulator. The ECG Simulator can be powered by just about any wall wart (DC power supply) you have around (for example, a 12 volt 500 ma supply). Now all we need is some software!

Figure 46: Completed ECG Simulator

Software Development

While the Arduino script required to implement the ECG waveform generation is not difficult, it is a little more involved than the "*blink an LED*" introductory examples one may encounter. Here are some of the issues one must think about before entering the first line of code.

Waveform as a C-language Array

This project has two important criteria concerning the waveform. The sample rate will be 1000 times a second (1 millisecond per sample); this was chosen for good waveform fidelity. The scale for this waveform should be 0 to 4095 (to make full-scale use of a 12-bit unipolar D/A converter).

The waveform should reside in EPROM, easily satisfied by using a const C-array with an initializer in the following form:

```
const short  y_data[] = {
                           939, 940, 941, 942, 944, 945, 946, 947, 951, 956,
                           962, 967, 973, 978, 983, 989, 994, 1000, 1005, 1015
                         };
```

By declaring the waveform array as a const array, it is assembled in the 32k EPROM rather than the more precious 2k of RAM which will be needed for variables, etc.

Waveform Updated as a Timer2 Interrupt

The waveform will be output at an update rate of 1000 samples per second. To do this, Timer2 will be used to count out a one millisecond period and then trigger a Timer2 interrupt. Within the interrupt routine, the next sample will be moved from the stored waveform array and sent to the D/A converter via the SPI interface. The Timer2 is restarted and this continues ad-infinitum (forever).

Heart Rate Displayed every 50 Milliseconds

Every 50th entry to the Timer2 interrupt routine, the heart rate selected by the user via the potentiometer will be sent to the 4-digit numeric display (again using the SPI interface). Both the D/A and the 4-digit display are updated within the Timer2 interrupt routine. Since they are serialized (one after the other), there will be no contention on the SPI interface bus.

Heart Rate Pot Read in the Background Loop

The Arduino background loop is where the analog voltage (0 to 5 volts) set by the pot is read using an analog input. This value will be used to specify the number of samples in the "quiescent period" of the waveform, specifically that flat-line period after the T-wave that continues to the start of the next PQRS complex. This "quiescent period" value will be written to a variable that will be read by the Timer2 interrupt routine.

The first job is to find and digitize a suitable ECG waveform and convert it into the C-language array construct mentioned previously.

Screen Capture a Suitable ECG Waveform

To get started, we need a suitable ECG waveform to digitize. The obvious thing to do is to find a waveform on the Internet and do a screen capture (convert it into a jpeg image file). A very good screen capture utility is MWSnap; it's free and easy to use. Refer to Appendix 1 for detailed instructions on downloading, installing, and using MWSnap. If you are running Windows 7 or 8, there's a nice accessory called the “snipping tool” that can do the same job. You'll find it in the start menu under “accessories”.

The waveform selected, shown in Figure 47, is from an Army Flight Medical Training Course on *Understanding ECG Waveforms.* (http://www.cs.amedd.army.mil/FileDownloadpublic.aspx?docid=f01e24ca-4afc-4d90-9d7f-aa7b693a269e). This waveform was chosen for three reasons, it was relatively uncluttered, had both axes labeled numerically, and being a government document there should be no copyright violation in using it. I used the MWSnap screen capture utility to grab this image and create a jpeg file. The jpeg image was then converted to black-and-white and sharpened via the open source Picasa photo editor from Google.

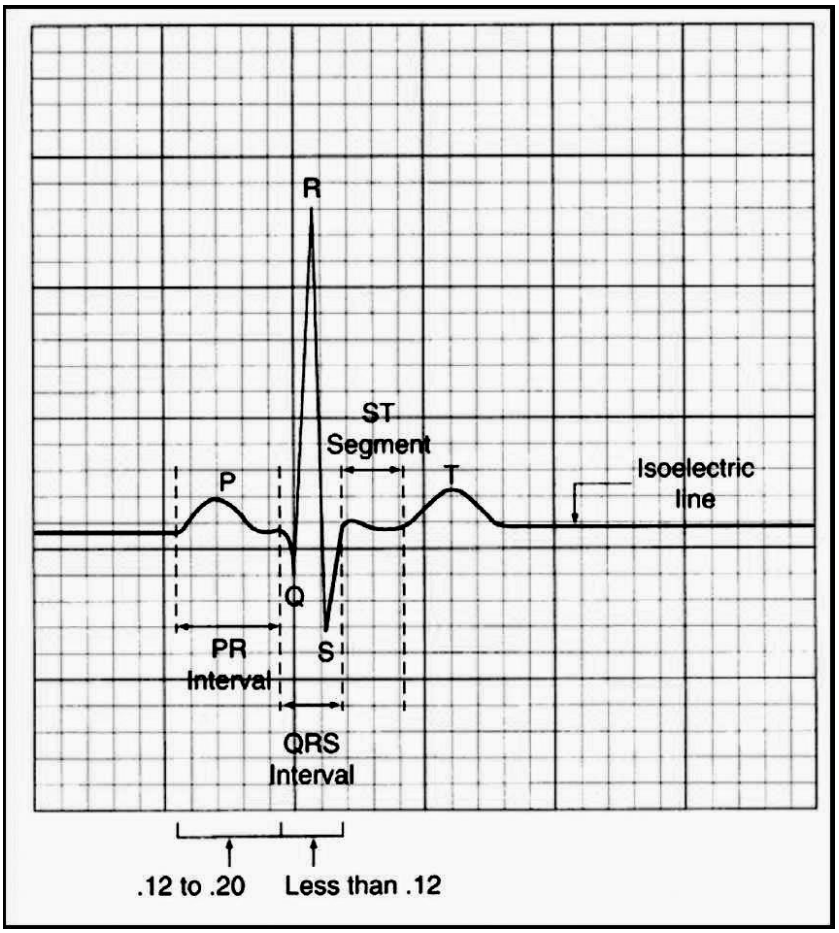


Figure 47: Sample ECG Waveform Found on the Internet

The screen captured ECG waveform was named “**SampledECGWaveform.jpg**” and stored in my **c:\temp** folder. The typical heart waveform is composed of the P-wave, the QRS complex (you know, the “ba-bump” part of your heart beat), and the T-wave.

On the horizontal time axis, major grid lines are at 0.200 seconds and the smallest grid divisions are 0.040 seconds. On the vertical amplitude axis, the major divisions are 1.0 millivolts and the smallest vertical divisions are 0.200 millivolts. After the last data point of the T-wave in Figure 47, the ECG waveform is quiescent until the next PQRST complex. That quiescent period (holding a constant sample) can be altered to vary the heart rate.

Normal heart rate for most people is 60 beats per minute or, in other words, the R-Wave peaks are one second apart. Now it's true that as the heart rate really speeds up, the QRS waveform compresses somewhat, but we will simplify the simulator by outputting the same QRS part followed by a variable quiescent part.

The other thing to know is that the amplitude of the ECG as measured by attached electrodes on the skin is just a couple of millivolts. We will now digitize the waveform in Figure 47 with the given time and amplitude axes using the Open Source digitizer program Engauge.

Installing and Using the Engauge Digitizer

The Open Source digitizer Engauge can be used to digitize (convert to numbers) the ECG waveform we previously captured. Engauge is hosted by SourceForge, so go to the SourceForge website (http://www.sourceforge.net/) and search for Engauge, as shown in Figure 48 below.

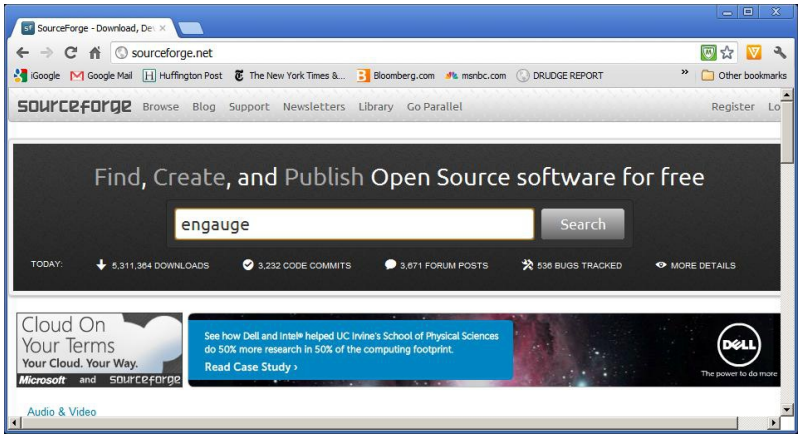


Figure 48: Go to www.sourceforge.net and search for Engauge

Engauge is supplied as a zip file ready to be copied to your hard drive, Download and unzip the contents into a folder on your C drive such as **c:\engauge**.

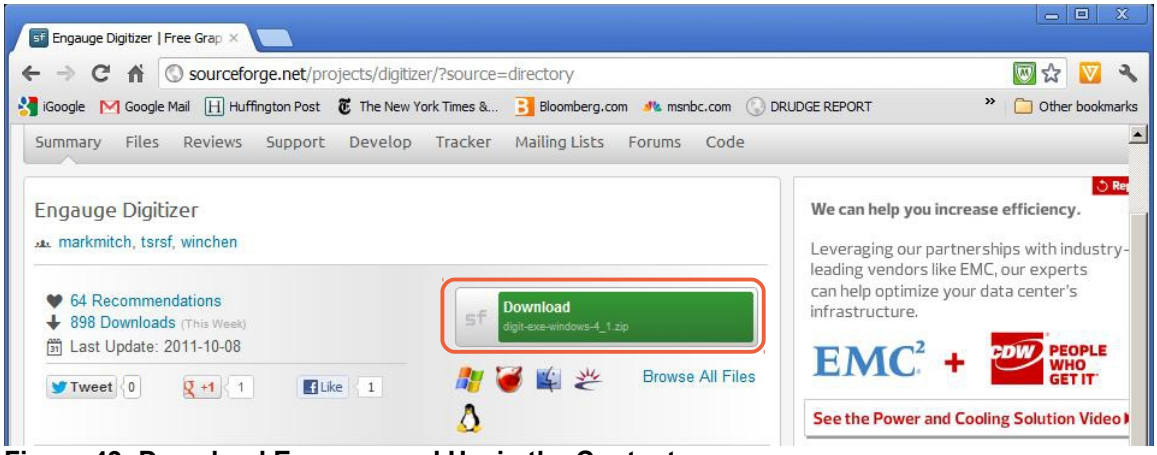


Figure 49: Download Engauge and Unzip the Contents

Create a desktop icon of the **engauge.exe** application and start it.

Engauge will start and present the opening screen shown in Figure 50 below.

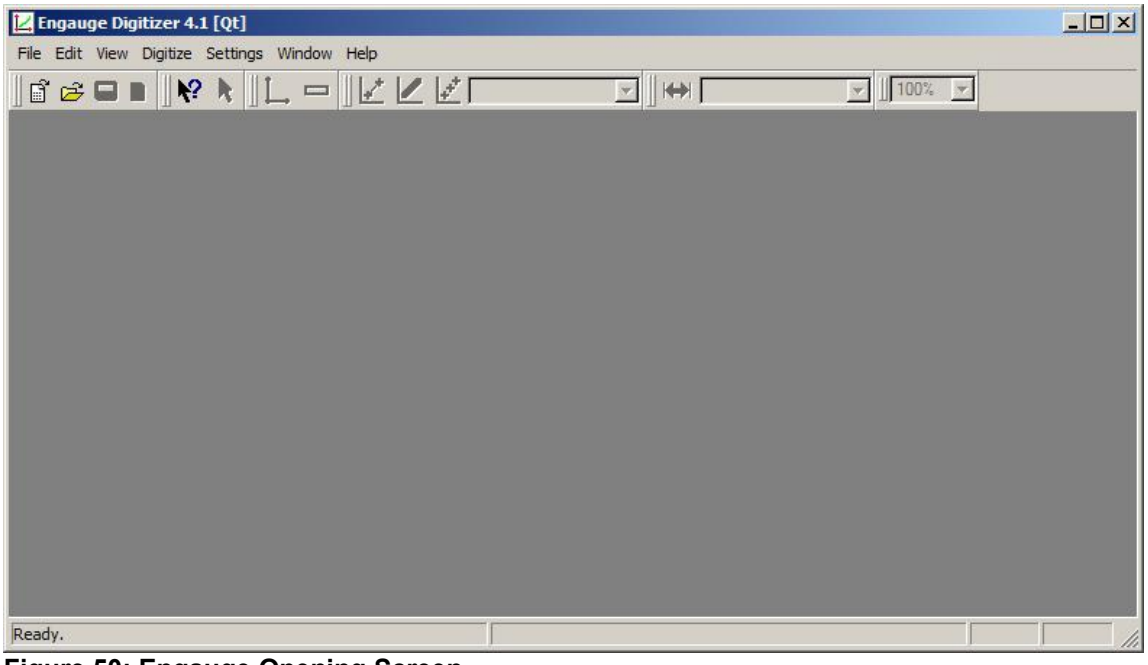


Figure 50: Engauge Opening Screen

Under the “**File**” pull-down menu, select **Import** as shown in Figure 51 below.

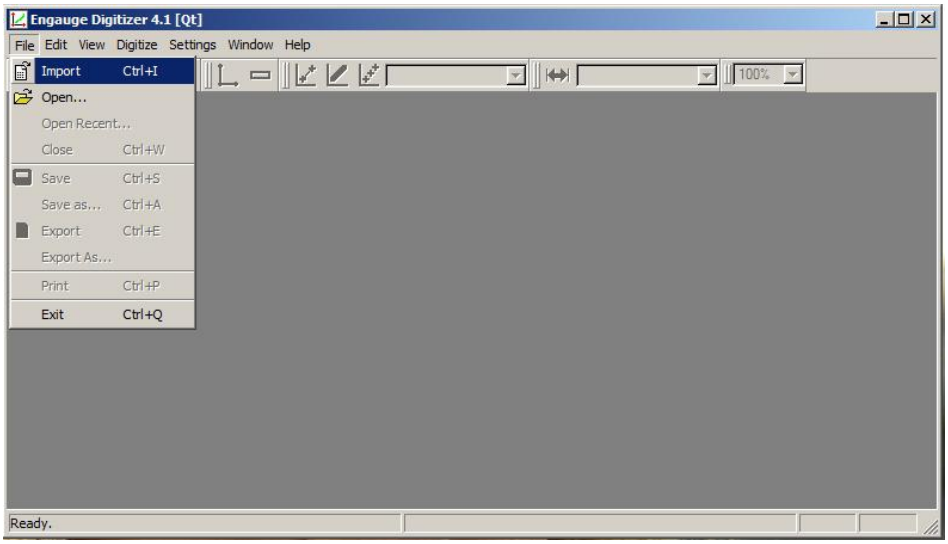


Figure 51: Click "File - Import" to Import our ECG Waveform

The “Import” menu presents a standard Windows file access form and for this example we will access the folder “**c:\temp**” and import the sampled ECG waveform “**SampledECGWaveform.jpeg**” as shown in Figure 52.

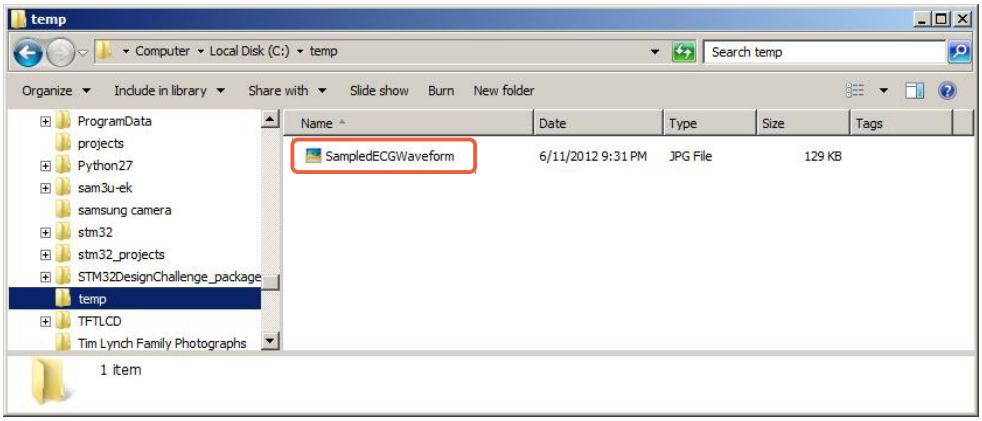


Figure 52: Importing the ECG waveform jpeg file

Now Engauge presents the jpeg waveform captured from the Internet, as shown in Figure 53 below. The three axes points that will be used for calibration in Figure 53 were added by the author for clarity, Engauge does not show these points. The opening Engauge screen will also show some colors indicating that it has detected the grid lines and trace.

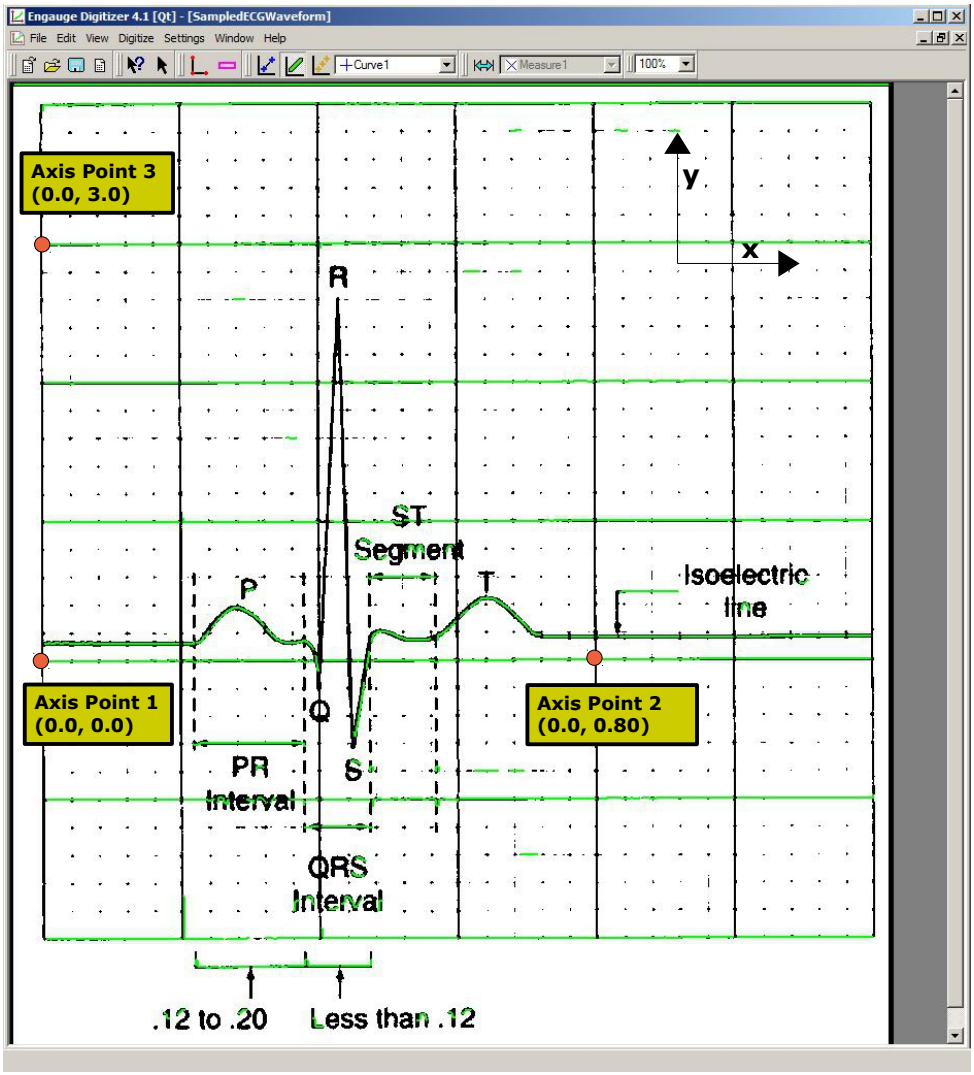


Figure 53: Engauge Has Imported our Waveform

Engauge requires three axis points be marked to define all the dimensions. Click on the **Axis Definition** tool as shown in Figure 54.

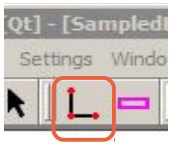


Figure 54: Axis Definition Tool

Note that in Engauge, the vertical axis is Y (amplitude) and the horizontal axis is X (time). The scaling is 1.0 millivolt per major division in the Y-axis and 0.2 seconds per major division in the X-axis (from the Army document).

The procedure is to click the axis point and then fill out the data entry text form that pops up. Clicking "**OK**" completes the definition of the axis point.

Now in sequence, enter the origin by first clicking on the origin point labeled "***axis point 1***" on Figure 55. This pops-up a short form shown in Figure 55 and you can enter the numeric values for x and y (**0.0, 0.0**) followed by clicking "**OK**".

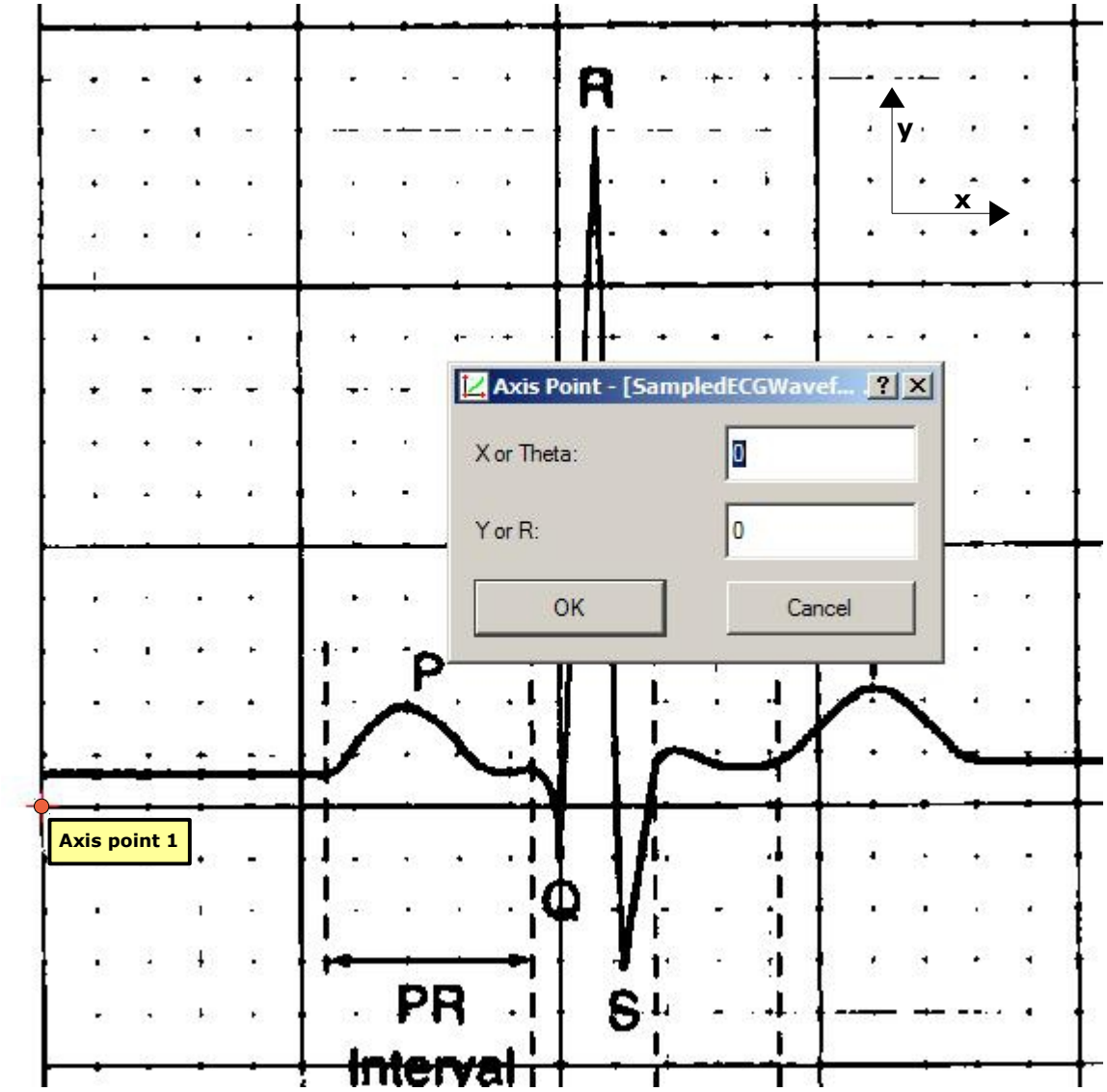


Figure 55: Entering Axis Point 1

Now click on the X-axis point labeled as "***axis point 2***" in Figure 56 below. When the data entry form appears, enter the numeric values for that point (0.0, 0.80) as shown in Figure 56.

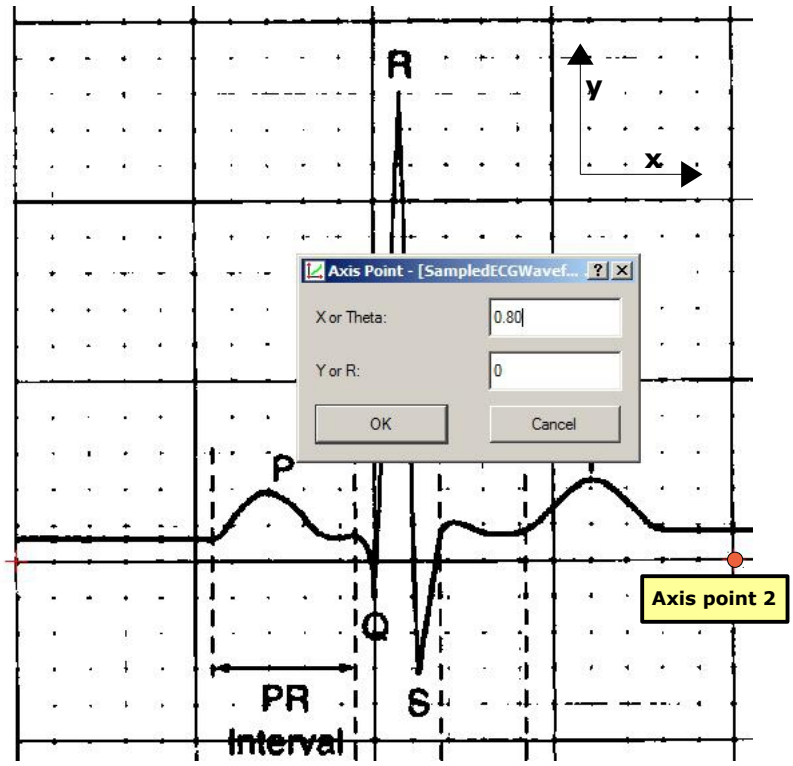


Figure 56: Entering Axis Point 2

Now click on the Y-axis point labeled as "***axis point 3***" on Figure 57 below. When the data entry form appears, enter the numeric values for that point (0.0, 3.0) as shown in Figure 57.

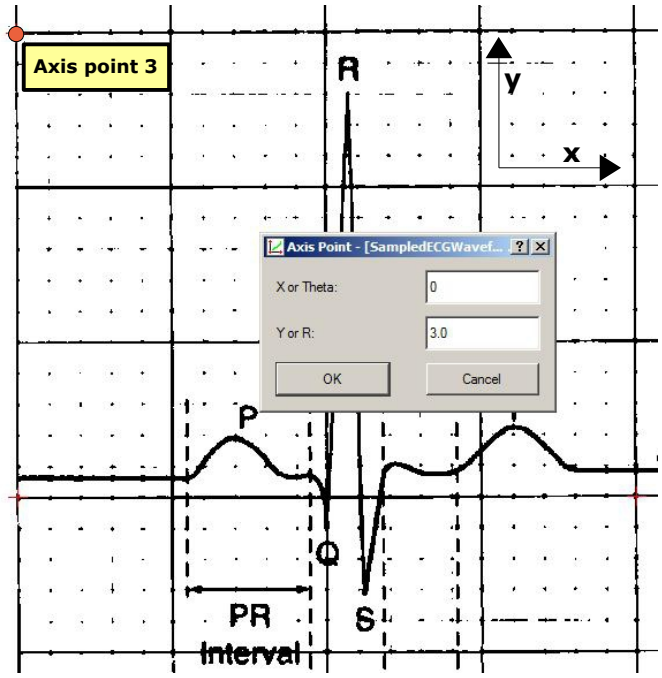


Figure 57: Entering Axis Point 3

Now we're ready to actually digitize points on the ECG curve. Click the “curve point” tool to start.

The procedure is pretty simple: you just click as many points on the curve as you can starting from left to right, as shown in Figure 58. Observe that we started at the beginning of the P-wave, not the origin point.

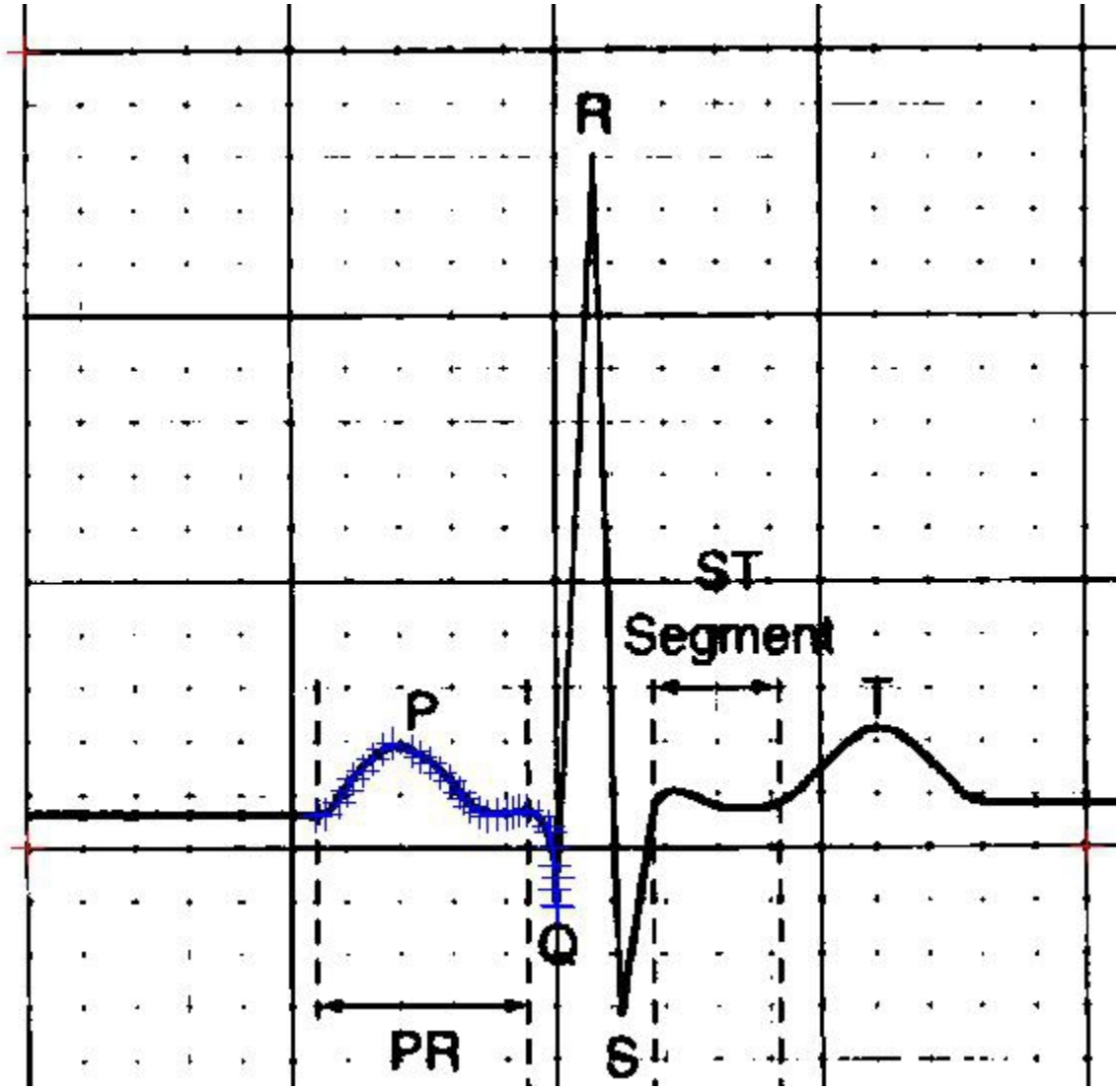


Figure 58: Clicking points on the curve (partial)

Obviously, the closer the clicked points are to each other, the better the finished product will be. Also, the points will not be evenly spaced so don't worry about that. We'll write a small Python program to use linear interpolation to make the points one millisecond apart.

It's also possible that the points will not necessarily be sequential (you accidentally clicked a point to the left of the last point you clicked). This will be also solved by the Python post-processing program that will sort the points so they are truly sequential. When you've clicked all along the curve, it will look like Figure 59. It took the author about five minutes to click all those points. We only want to digitize the PQRST points, not the “flat-line” segments.

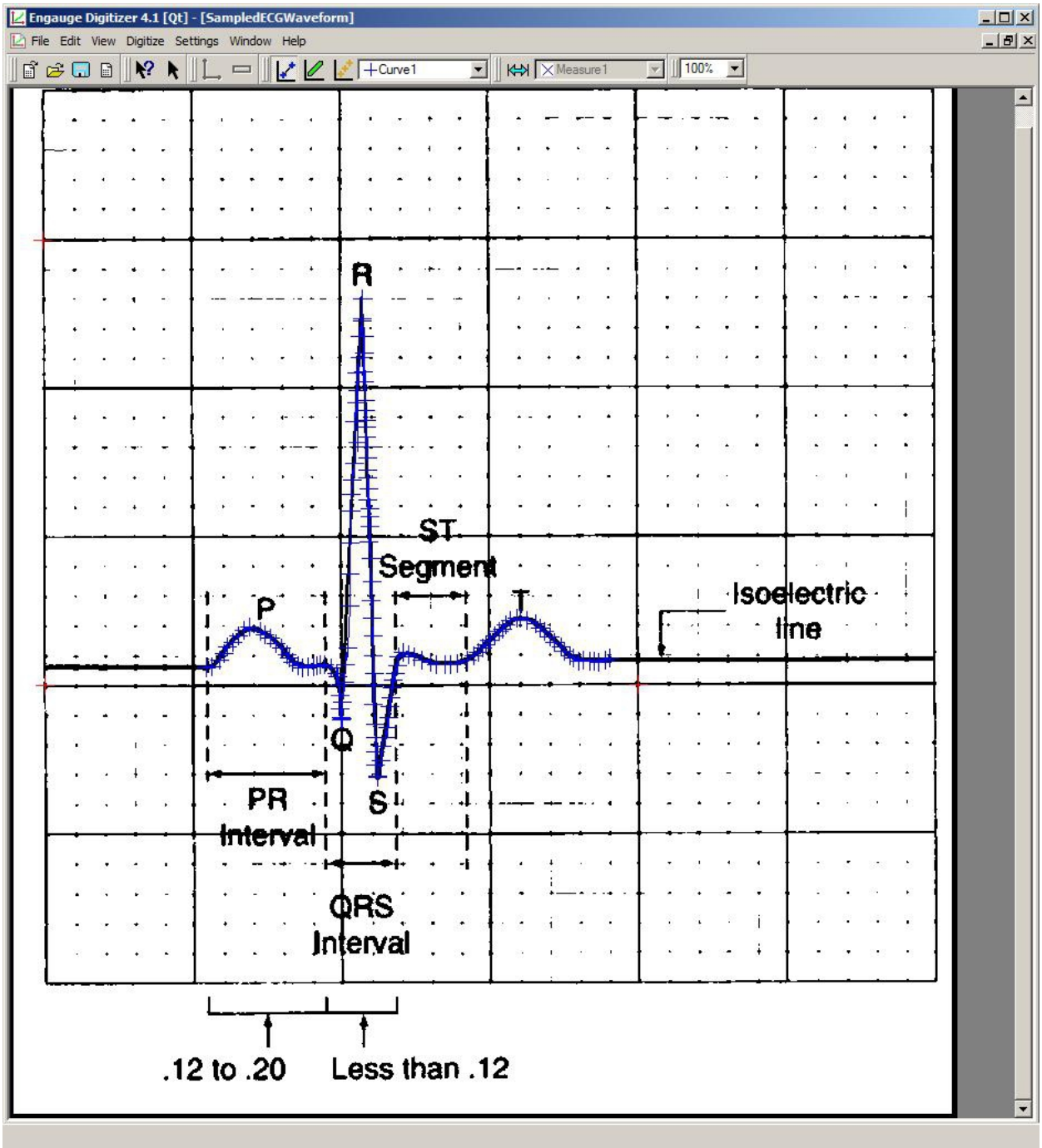


Figure 59: ECG Waveform Fully Digitized

The points in Figure 59 are in a tabular format as (x, y) pairs. These can be exported to a text file. Click “File – Export As...” as shown in Figure 60.

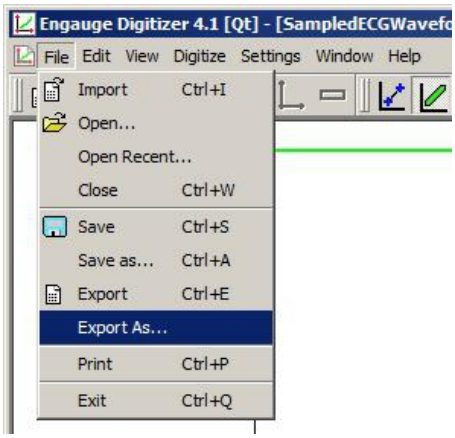


Figure 60: Saving to a Text File

The “Export” command will present a standard Windows file form and you can browse to the desired folder to hold the resulting text file plus give it any name you want. The output file extension should be **.txt** in any case, as shown in Figure 61 below. For this example, I gave the output file the name “**SampledECGWaveform.txt**”.

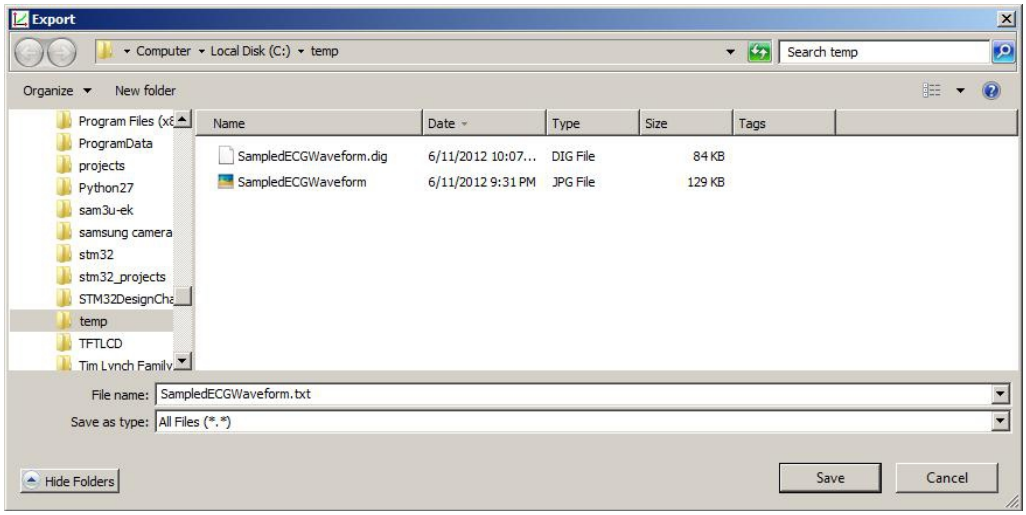


Figure 61: Specifying the Output File Name

If you use the Windows Notepad editor to inspect the output text file, it will resemble the text file shown in Figure 62. Note that the coordinates are scaled as per the original screen capture. The very first line of the text file (x,Curve1) will be removed by the Python post processing program so the resulting file will only consist of coordinate pairs.

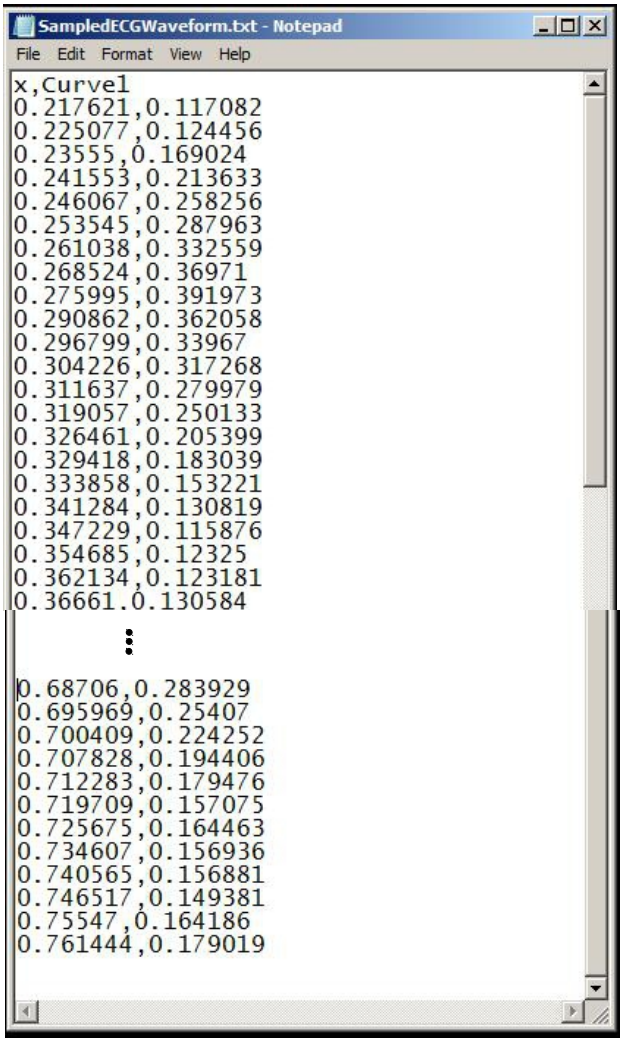


Figure 62: Digitized Data as a Text File

So we've digitized our ECG waveform that was screen-captured from the Internet and now have a text file called *SampledECGWaveform.txt***.** This file is composed of one data point per line where the data point is an (x, y) pair. To place this data in an Arduino C/C++ language sketch (program), a C language array of y-data points with an initializer can be created, as follows:

```
const short  y_data[ ] = {
    879, 879, 879, 879, 879, 879, 880, 885, 890, 895,
    900, 924, 961, 1006, 1043, 1080, 1122, 1179, 1207, 1235,
                  :
    1000, 984, 966, 947, 929, 913, 901, 892, 892, 890,
    883, 878, 877, 875, 874, 872, 874, 879 };
```

The Y-data points are scaled to the D/A converter chosen. Since we will be using a Microchip MCP4921 12-bit Digital to Analog (D/A) converter, the Y-values will be scaled between 0 and 4095. By deciding at the outset to have the Arduino output a point from this table every one millisecond, there is no need to store any X-values in the Arduino.

Python Post Processing Program

At this point we have a text file with the waveform's (x, y) data points and now wish to convert this to a C language array with intitializer. There are a few problems to overcome.

- The digitized data points range from 0.217621 to 0.761444 seconds in the x-axis (time) and from -0.622033 to 2.60146 in the y-axis (amplitude). Our final scaling will be 0.0 to 543.0 milliseconds in the x-axis (time) and 0 to 4095 units in the y-axis (amplitude). The amplitude scaling reflects the full-range use of a unipolar 12-bit digital-to-analog converter (D/A).

- The horizontal (x-axis) points are not evenly spaced. We want these to be 1.0 msec apart. That's our sample rate and will give a very detailed rendering of our waveform.

- Some data points may not be sequential (you may have clicked a point to the left of the last point entered).

A program was written to read the data file from the Engauge digitizer and create an output text file that looks like a C language array definition with a initializer. To build this program, I chose the *Python* language. The reasons for choosing *Python* are that it's Open Source and free, the language is especially good at manipulating lists, and there's lots of free documentation on the Web.

You need to download two things: the *Python* language installer and the PYScripter IDE which provides a very nice editor/debugger.

The *Python* language can be downloaded from here: http://www.python.org/getit/

The PyScripter IDE can be downloaded from here: http://code.google.com/p/pyscripter/downloads/list

With both of these downloads, you can choose either a Windows 32-bit or a Windows 64-bit version. There are also other versions for the Mac and Linux platforms. The *Python* language is offered in two main revisions, version 2.7.3 and the latest version 3.2.3. The *Python* post processing program was built with the earlier version, version 2.7.3 so it's recommended that you do the same. Sadly, one weakness of *Python* is that the revisions have not always been backward-compatible. There are a truckload of *Python* libraries available, all open source and free. The only problem is that they need time to “catch up” with the newest *Python* revision.

Since *Python* is an interpreter, you must download and install *Python* 2.7.3 to run the post processing program. The post-processing *Python* application is called:

create_ecg_data.py

The Python post-processing program (create_ecg_data.py) and the captured ecg waveform jpg file (*SampledECGWaveform.txt*) was placed into a folder **c:\projects**.

To execute the post-processing program, there are two choices. You can start the Python built-in IDE called IDLE, locate the post-processing program, and execute it. Alternatively, you could be more sophisticated and use the PYScripter IDE which has a better editor and a very nice debugger.

The Python program creates, as its output, a file named *ecg_ydata_dtoa.txt*. This is a simple text file with the C Language const array with initializer that specifies the ecg waveform data points.

What follows directly below is a complete listing of the Python post-processing program (**create_ecg_data.py**). The source code for this Python application is hosted at GitHub (search for lynchzilla).

PYTHON POST PROCESSING PROGRAM

```
#-------------------------------------------------------------------------------
# Name:       create_ecg_data
# Purpose:    create ecg waveform data in C language format
#
# Author:     Jim Lynch
# Language:   Python  2.7.3
# Created:    22/07/2012
# Copyright:  (c) Jim 2012
# Licence:    <Open Source>
#
# Note: tested with Python 2.7.3
#-------------------------------------------------------------------------------
#!/usr/bin/env python
import operator
def main():
   pass
if __name__ == '__main__':
   main()

   ###############################################################################################
   #          interpolate(x, x_values, y_values)
   #
   #  Description:  return y at point x using linear interpolation
   #
   #  Input:     x = input x-value
   #             x_values = table of digitized points along horizontal x-axis (time)
   #             y_values = table of digitized points along vertical y-axis (amplitude)
   #
   #
   #  Returns:  y = y-value at the point x
   #
   #  Reference:   Dr. Dogan Ibrahim
   #                  "Interpolation in microcomputer based instrumentation and Control Projects"
   ###############################################################################################
   def interpolate(x, x_values, y_values):

      # find the table points on either side of the input value x
      # note: a is the x-index before and b is the x-index after
      i = 0
      while (x > x_values[i]):
         i = i + 1

      # xa = x value before the input x
      # xb = x value after the input x
      xa = x_values[i - 1]
      xb = x_values[i]

      # fa = y value at xa
      # fb = y value at xb
      fa = y_values[i - 1]
      fb = y_values[i]

      # the linear interpolation formula
      y1 = fa * (xb - x) / (xb - xa)
      y2 = fb * (x - xa) / (xb - xa)
      y = y1 + y2

      # return the y value at point x
      return y
```

```
#-------------------------------------------------------------------------------------------
#               WAVEFORM GENERATOR UTILITY
#
#
#  Purpose: Create a table of ECG waveform points, spaced every 1.0 msec
#
#
#  Input:   Digitized waveform from Engauge (open source, digitizing software)
#             Engauge may be download from SourceForge
#
#
#           File: SampledECGWaveform.txt has x,y values (1 point per line)
#
#           x,Curve1          <====== THIS LINE WILL BE REMOVED
#           0.217621,0.117082
#           0.225077,0.124456
#           0.23555,0.169024
#           0.241553,0.213633
#           0.246067,0.258256
#           0.253545,0.287963
#           0.261038,0.332559
#           0.268524,0.36971
#             :      :
#           0.746517,0.149381
#           0.75547,0.164186
#           0.761444,0.179019
#
#  Processing:   Data points are read from above file and using linear
#                 interpolation are converted into samples spaced every 1 msec
#
#                 y-axis data is scaled to 0 .. 4096
#                 (for Microchip MCP4921-E/P-ND 12-bit D/A converter)
#
#
#  Output:  text file contains single table of y-values spaced every 1 msec
#             file is formatted to look like a C array with initialization.
#
#     const short  y_data[] = {
#     936, 938, 939, 940, 941, 943, 944, 945, 949, 954,
#     959, 965, 970, 976, 981, 986, 992, 997, 1003, 1012,
#     1022, 1031, 1041, 1050, 1060, 1072, 1085, 1097, 1110, 1118,
#     1123, 1128, 1133, 1138, 1143, 1148, 1154, 1161, 1169, 1176,
#     1184, 1191, 1199, 1206, 1213, 1219, 1226, 1232, 1238, 1245,
#     1251, 1257, 1261, 1265, 1268, 1272, 1276, 1280, 1283, 1283,
#     1281, 1278, 1276, 1273, 1270, 1268, 1265, 1263, 1260, 1258,
#     1255, 1253, 1250, 1248, 1243, 1238, 1234, 1229, 1224, 1219,
#     1215, 1212, 1208, 1204, 1200, 1196, 1192, 1188, 1181, 1175,
#     1168, 1162, 1156, 1149, 1143, 1138, 1133, 1128, 1123, 1117,
#     1112, 1107, 1101, 1093, 1085, 1078, 1070, 1062, 1055, 1047,
#     1037, 1028, 1018, 1010, 1001, 993, 984, 979, 975, 972,
#     968, 964, 960, 956, 953, 950, 946, 943, 940, 937,
#     935, 937, 938, 939, 940, 942, 943, 944, 944, 944,
#     944, 944, 944, 944, 944, 945, 947, 949, 951, 954,
#     956, 958, 960, 962, 963, 963, 963, 963, 963, 963,
#     961, 958, 955, 951, 948, 945, 942, 939, 935, 930,
#     924, 917, 911, 905, 898, 892, 882, 863, 818, 731,
```

```
#      604, 553, 506, 630, 696, 750, 805, 894, 975, 1021,
#      1066, 1124, 1234, 1344, 1454, 2080, 2241, 2468, 2543, 2588,
#      2634, 2689, 3076, 3127, 3179, 3209, 3307, 3395, 3484, 3572,
#      3795, 3838, 3881, 3789, 3434, 3444, 3289, 3045, 2812, 2803,
#      2220, 2252, 1888, 1730, 1621, 995, 901, 354, 375, 203,
#      30, 32, 61, 90, 119, 160, 237, 275, 291, 308,
#      325, 343, 370, 398, 428, 483, 541, 601, 650, 701,
#      756, 800, 836, 854, 873, 893, 914, 936, 965, 1014,
#      1032, 1038, 1045, 1051, 1057, 1064, 1064, 1061, 1058, 1056,
#      1053, 1051, 1048, 1046, 1043, 1041, 1038, 1035, 1033, 1030,
#      1028, 1025, 1022, 1020, 1017, 1014, 1011, 1009, 1006, 1003,
#      1001, 998, 997, 995, 993, 992, 990, 989, 987, 986,
#      984, 982, 981, 979, 976, 974, 971, 968, 966, 963,
#      961, 961, 961, 961, 961, 961, 961, 961, 961, 961,
#      961, 961, 961, 961, 961, 961, 961, 961, 961, 961,
#      962, 963, 964, 965, 966, 967, 968, 969, 970, 972,
#      974, 976, 978, 980, 982, 984, 986, 989, 991, 993,
#      995, 997, 1000, 1004, 1008, 1012, 1017, 1021, 1025, 1029,
#      1033, 1038, 1043, 1047, 1052, 1057, 1061, 1067, 1073, 1079,
#      1086, 1092, 1098, 1105, 1111, 1117, 1123, 1130, 1138, 1146,
#      1155, 1163, 1170, 1175, 1180, 1185, 1190, 1195, 1200, 1205,
#      1211, 1218, 1224, 1230, 1237, 1243, 1247, 1251, 1256, 1260,
#      1266, 1274, 1283, 1291, 1300, 1306, 1312, 1318, 1325, 1331,
#      1337, 1339, 1341, 1344, 1346, 1348, 1350, 1352, 1354, 1356,
#      1356, 1356, 1356, 1356, 1355, 1353, 1351, 1348, 1346, 1344,
#      1342, 1340, 1338, 1335, 1333, 1331, 1328, 1326, 1324, 1321,
#      1319, 1316, 1314, 1312, 1309, 1304, 1297, 1291, 1285, 1278,
#      1272, 1267, 1262, 1257, 1252, 1248, 1243, 1237, 1230, 1224,
#      1218, 1211, 1205, 1198, 1191, 1183, 1175, 1167, 1159, 1151,
#      1145, 1141, 1137, 1133, 1128, 1124, 1120, 1116, 1111, 1104,
#      1096, 1087, 1079, 1071, 1066, 1061, 1056, 1051, 1046, 1041,
#      1035, 1031, 1027, 1023, 1018, 1014, 1010, 1007, 1003, 999,
#      995, 991, 987, 989, 990, 992, 993, 995, 996, 995,
#      994, 993, 992, 991, 990, 989, 988, 987, 987, 987,
#      987, 987, 987, 987, 985, 984, 982, 980, 979, 978,
#      980, 982, 984, 986, 988, 990, 992, 994, 997, 1000,
#      1003, 1006, 1009};
#
#        This text file can be easily cut-and-pasted into an Arduino sketch.
#
#--------------------------------------------------------------------------------------------

# create some empty lists
ecg_pt_x = []
ecg_pt_y = []
ecg_x = []
ecg_y = []
ecgx = []
ecgy = []
ecg_xdata_ms = []
ecg_ydata_dtoa = []
```

```
# open the waveform text file (from engauge)
# note: waveform file should be in the same folder as the python program (c:\projects\)
file = open("SampledECGWaveform.txt")

# read and discard the first line of the file  ( x,Curve1 )
line = file.readline()

# endless loop to read and process the waveform file
while 1:
   # read a line from the raw waveform file
   # note: each line in the file is a string: '0.297647,0.321337\n'
   line = file.readline()

   # break (exit from endless loop) when there are no more lines to read
   if not line:
      break

   # process the file
   pass


   # remove the '\n' newline character
   # line = '0.297647,0.321337'
   line = line.rstrip('\n')

   # split the string into a list at the comma separator
   # linelist = ['0.297647', '0.321337']
   linelist = line.split(',')

   # now extract the x-value and y-value and convert to float
   x = float(linelist[0])
   y = float(linelist[1])

   # add these x and y values to the respective point lists
   # ecg_pt_x = [0.297647, 4.45898, 9.21484] ... and so on
   # ecg_pt_y = [0.321337, 0.0, -0.321337]   ... and so on
   ecg_pt_x.append(x)
   ecg_pt_y.append(y)

# note the size of each list
lengthx = len(ecg_pt_x)
lengthy = len(ecg_pt_y)

# note the start and end points of the lists
xstart = ecg_pt_x[0]
ystart = ecg_pt_y[0]
xend = ecg_pt_x[lengthx - 1]
yend = ecg_pt_y[lengthy - 1]

# x-axis is horizontal (time)
# total time from start to finish is about 434 milliseconds
```

```
# x_axis: adjust x-axis so it starts at zero
for i in range(lengthx):
   if (xstart < 0):
      ecg_pt_x[i] = ecg_pt_x[i] + xstart
   else:
      ecg_pt_x[i] = ecg_pt_x[i] - xstart

# recalculate the start/end points
xstart = ecg_pt_x[0]
xend = ecg_pt_x[lengthx - 1]
xspan = xend - xstart

# y-axis is vertical  (amplitude)
# total span from min to max should be 0.0 .. 4096.0
# (Microchip MCP4921-E/P-ND 12-bit D/A - unipolar)

# calculate the y-axis max and min values
ymin = min(ecg_pt_y)
ymax = max(ecg_pt_y)

# adjust y-axis so it's minimum is zero
for i in range(lengthy):
   if (ymin < 0):
      ecg_pt_y[i] = ecg_pt_y[i] - ymin
   else:
      ecg_pt_y[i] = ecg_pt_y[i] + ymin

# calculate the y-axis span
yspan = ymax - ymin
yscalefactor = 4096 / yspan

# scale y-axis to 0 .. 4096
for i in range(lengthy):
   ecg_pt_y[i] = ecg_pt_y[i] * yscalefactor

# convert ecg_pt_x and ecg_pt_y lists to a list of tuples
ecg_data = list(zip(ecg_pt_x, ecg_pt_y))

# sort list of tuples by ecg_pt_x (Time axis)
ecg_data.sort(key=operator.itemgetter(0))

# extract list of x-values and list of y-values (both same size)
for i in range(len(ecg_data)):
   ecg_x.append(ecg_data[i][0])
   ecg_y.append(ecg_data[i][1])

# convert x-values into milliseconds
for i in range(len(ecg_data)):
   ecg_x[i] = ecg_x[i] * 1000.0
```

```
# create ecg_xdata[] and ecg_ydata[] tables at 1 msec intervals
x = 0.0
for i in range(int(ecg_x[-1])):

   # linear interpolate every 1.0 msec
   y = interpolate(x, ecg_x, ecg_y)

   # add to ecg_xdata_ms[] and ecg_ydata_dtoa[] tables
   ecg_xdata_ms.append(int(x))
   ecg_ydata_dtoa.append(int(y))
   x = x + 1.000

# create a file to write the ecg_ydata to
FILE = open("ecg_ydata_dtoa.txt","w")

# create a C language array definition of the y-values, 10 values per line
j = 0
string = "const short  y_data[] = {\n"
FILE.write(string)
string = ''

for i in range(int(ecg_x[-1])):

   string = string + str(ecg_ydata_dtoa[i]) + ', '
   j = j + 1

   # reached the last element?
   if (i == int(ecg_x[-1]) - 1):

      # reached the last element, must close line with a brace
      # convert string into a list
      liststring = list(string)

      # remove trailing space and comma
      del liststring[-1]
      del liststring[-1]

      # convert back to string
      string = ''.join(liststring)

      # write final string to file with closing brace and semicolon
      FILE.write(string + '};\n')

  else:
      # write next line to file
      if (j >= 10):
         FILE.write(string + '\n')
         string = ''
         j = 0

# all done, close file "ecg_ydata_dtoa.txt"
# (it may be used in Arduino waveform generator program)
FILE.close()
```

Utilizing Python 2.7.3 and executing the **create_ecg_data.py** post-processing program, an output file called **ecg_ydata_dtoa.txt** will be created that has the data points formatted as a C language array with initializers, as shown below in Figure 63.

File: ecg_ydata_dtoa.txt

```
const short  y_data[ ] = {
 939, 940, 941, 942, 944, 945, 946, 947, 951, 956,
 962, 967, 973, 978, 983, 989, 994, 1000, 1005, 1015,
 1024, 1034, 1043, 1053, 1062, 1075, 1087, 1100, 1112, 1121,
 1126, 1131, 1136, 1141, 1146, 1151, 1156, 1164, 1172, 1179,
 1187, 1194, 1202, 1209, 1216, 1222, 1229, 1235, 1241, 1248,
 1254, 1260, 1264, 1268, 1271, 1275, 1279, 1283, 1287, 1286,
 1284, 1281, 1279, 1276, 1274, 1271, 1268, 1266, 1263, 1261,
 1258, 1256, 1253, 1251, 1246, 1242, 1237, 1232, 1227, 1222,
 1218, 1215, 1211, 1207, 1203, 1199, 1195, 1191, 1184, 1178,
 1171, 1165, 1159, 1152, 1146, 1141, 1136, 1130, 1125, 1120,
 1115, 1110, 1103, 1096, 1088, 1080, 1073, 1065, 1057, 1049,
 1040, 1030, 1021, 1012, 1004, 995, 987, 982, 978, 974,
 970, 966, 963, 959, 955, 952, 949, 945, 942, 939,
 938, 939, 940, 941, 943, 944, 945, 946, 946, 946,
 946, 946, 946, 946, 946, 947, 950, 952, 954, 956,
 958, 960, 962, 964, 965, 965, 965, 965, 965, 965,
 963, 960, 957, 954, 951, 947, 944, 941, 938, 932,
 926, 920, 913, 907, 901, 894, 885, 865, 820, 733,
 606, 555, 507, 632, 697, 752, 807, 896, 977, 1023,
 1069, 1127, 1237, 1347, 1457, 2085, 2246, 2474, 2549, 2595,
 2641, 2695, 3083, 3135, 3187, 3217, 3315, 3403, 3492, 3581,
 3804, 3847, 3890, 3798, 3443, 3453, 3297, 3053, 2819, 2810,
 2225, 2258, 1892, 1734, 1625, 998, 903, 355, 376, 203,
 30, 33, 61, 90, 119, 160, 238, 275, 292, 309,
 325, 343, 371, 399, 429, 484, 542, 602, 652, 703,
 758, 802, 838, 856, 875, 895, 917, 938, 967, 1016,
 1035, 1041, 1047, 1054, 1060, 1066, 1066, 1064, 1061, 1058,
 1056, 1053, 1051, 1048, 1046, 1043, 1041, 1038, 1035, 1033,
 1030, 1028, 1025, 1022, 1019, 1017, 1014, 1011, 1008, 1006,
 1003, 1001, 999, 998, 996, 994, 993, 991, 990, 988,
 986, 985, 983, 981, 978, 976, 973, 971, 968, 966,
 963, 963, 963, 963, 963, 963, 963, 963, 963, 963,
 963, 963, 963, 963, 963, 963, 963, 963, 963, 963,
 964, 965, 966, 967, 968, 969, 970, 971, 972, 974,
 976, 978, 980, 983, 985, 987, 989, 991, 993, 995,
 997, 999, 1002, 1006, 1011, 1015, 1019, 1023, 1028, 1032,
 1036, 1040, 1045, 1050, 1055, 1059, 1064, 1069, 1076, 1082,
 1088, 1095, 1101, 1107, 1114, 1120, 1126, 1132, 1141, 1149,
 1158, 1166, 1173, 1178, 1183, 1188, 1193, 1198, 1203, 1208,
 1214, 1221, 1227, 1233, 1240, 1246, 1250, 1254, 1259, 1263,
 1269, 1278, 1286, 1294, 1303, 1309, 1315, 1322, 1328, 1334,
 1341, 1343, 1345, 1347, 1349, 1351, 1353, 1355, 1357, 1359,
 1359, 1359, 1359, 1359, 1358, 1356, 1354, 1352, 1350, 1347,
 1345, 1343, 1341, 1339, 1336, 1334, 1332, 1329, 1327, 1324,
 1322, 1320, 1317, 1315, 1312, 1307, 1301, 1294, 1288, 1281,
 1275, 1270, 1265, 1260, 1256, 1251, 1246, 1240, 1233, 1227,
 1221, 1214, 1208, 1201, 1194, 1186, 1178, 1170, 1162, 1154,
 1148, 1144, 1140, 1136, 1131, 1127, 1123, 1118, 1114, 1107,
 1099, 1090, 1082, 1074, 1069, 1064, 1058, 1053, 1048, 1043,
 1038, 1034, 1029, 1025, 1021, 1017, 1013, 1009, 1005, 1001,
 997, 994, 990, 991, 992, 994, 996, 997, 999, 998,
 997, 996, 995, 994, 993, 991, 990, 989, 989, 989,
 989, 989, 989, 989, 988, 986, 984, 983, 981, 980,
 982, 984, 986, 988, 990, 993, 995, 997, 999, 1002,
 1005, 1008, 1012};
```

Figure 63: ECG Waveform Data Expressed as a C Language Array with Initializer

Store Only the PQRST Part of the Waveform

To save space in the Arduino, only the PQRST part of the heart waveform is stored. The “quiescent” part is just one sample emitted just once and thus held by the D/A converter at that value till the end of the “quiescent” period (the chosen “held” sample is very first sample in the PQRST waveform, 939).

Figure 64 shows a simple formula for determining the required number of “quiescent” samples for a specific BPM (beats per min) value. We can output rates from 30.0 BPM to 110.4 BPM with this scheme.

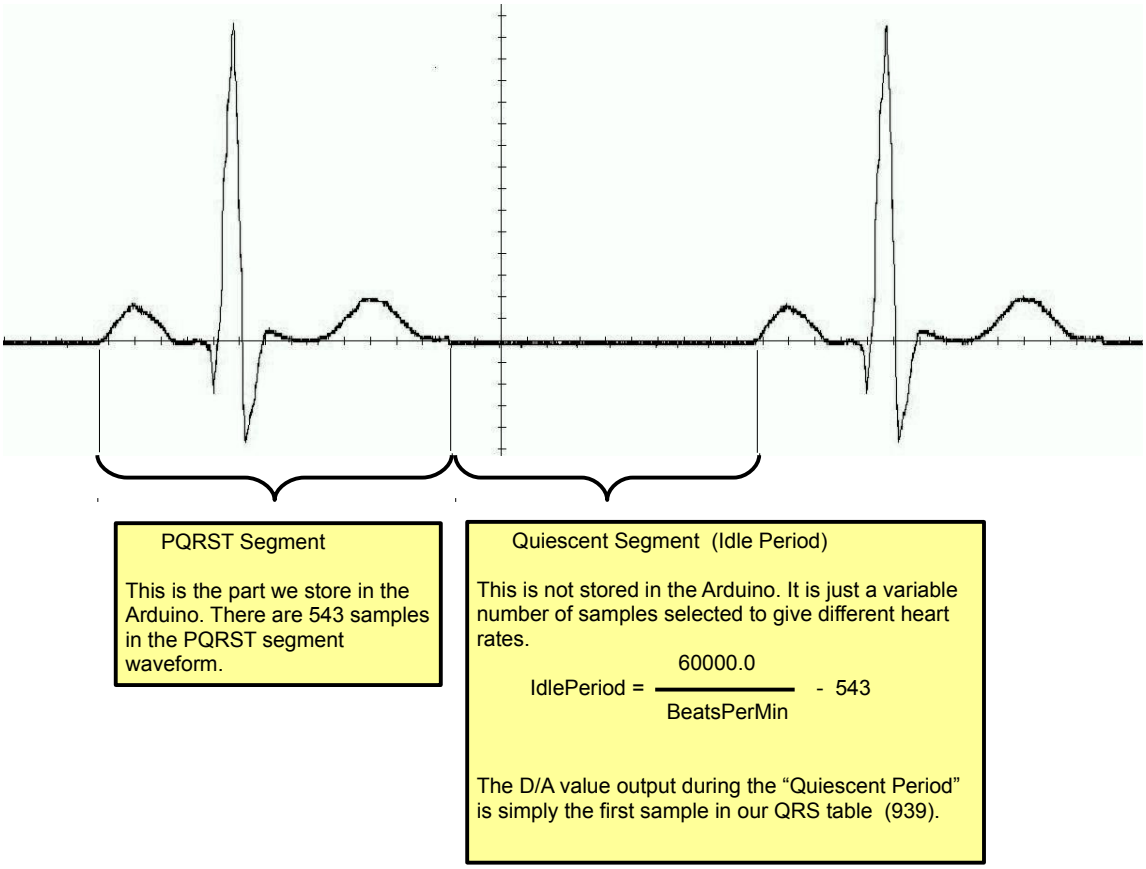


Figure 64: ECG waveform has two components

Create Arduino Sketch

Most readers probably know this, but Arduino requires two predefined functions, setup( ) and loop( ) shown in Figure 65. Normally, your job is to flesh out these two functions.

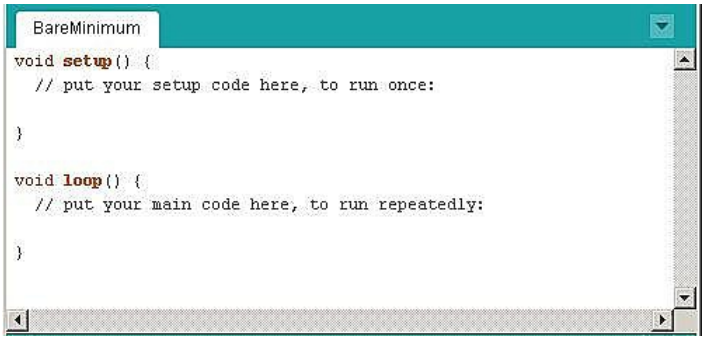


Figure 65: Basic Arduino Sketch Template

The setup( ) function contains statements that identify and initialize peripherals and ports you intend to use. The setup( ) function is run once and only once at boot-up.

The loop( ) function contains statements that actually execute your application. These statements are run, after the setup( ) function has completed, from top to bottom over and over again till hell freezes over (forever).

Let's look at some of the issues encountered in developing the Setup( ) function.

Set Up the SPI Interface

Both the D/A converter and the 4-digit numeric display use (and thus share) the SPI interface. SPI is a clock/data interface fully supported by the Arduino library (SPI.h). Australian blogger John Boxall has a nice tutorial on the SPI interface, you can find it here:

http://tronixstuff.wordpress.com/2011/06/15/tutorial-arduino-and-the-spi-bus-part-ii/

The following code, adapted from John's tutorial, sets up an SPI interface to the Microchip MCP4921 D/A converter and the Sparkfun 4-digit 7-segment display.

```
#include "SPI.h"        // supports the SPI interface to the D/A converter and 7-segment display
#include <Wire.h>       // need the Wire library

// Configure the output ports   (7-segment display and D/A SPI support)
pinMode(9, OUTPUT);                 // 7-segment display chip select   (low to select chip)
pinMode(10, OUTPUT);                // D/A converter chip select       (low to select chip)
pinMode(11, OUTPUT);                // SDI data
pinMode(13, OUTPUT);                // SCK clock

// initial state of SPI interface
SPI.begin();                            // wake up the SPI bus.
SPI.setDataMode(0);                     // mode: CPHA=0, data captured on clock's rising edge (low→high)
SPI.setClockDivider(SPI_CLOCK_DIV64);   // system clock / 64
SPI.setBitOrder(MSBFIRST);              // bit 7 clocks out first
```

In the code fragment above, we assign two output ports for the chip selects and two output ports for the SPI data and clock signals. In this application, the SPI interface's clock and data lines are “daisy-chained” to the two peripherals (the D/A and the display) and the individual chip select signals are used to select one or the other for operation.

By setting the clock divider to SPI_CLOCK_DIV64, we get the system clock (16 Mhz) divided by 64, which is a SPI clock rate of 250 Khz (a 4 microsecond period). That's pretty fast; clocking out the 16

bits required by the D/A would only require 64 usec.

Two other SPI characteristics are setup by the above code fragment, data is captured on the rising edge of the clock signal and all data is clocked out of the SPI peripheral most-significant bit first.

Setting Up the Timer2 Interrupt

It is NOT a good idea to output the next D/A sample from within the Arduino background loop; this method is not synchronous, even if we use a delay(1) in an attempt to get 1.0 milliseconds of delay in the loop. No, the best way is to use the Arduino Timer2 programmed to count out precisely 1.000 milliseconds and then interrupt. The Timer2 interrupt routine will fetch the sample and output it to the D/A converter.

The Timer2 8-bit counter actually counts upwards, interrupting when it hits 255 and thus “overflows” to zero. By selecting the clock pre-scaler to divide by 128, we'll get a count rate of 8 usec per count for Timer2. So we need 125 Timer2 clock ticks to get 1.000 msec (.000008 * 125 = 0.001 sec)

Therefore, we have to pre-load the Timer2 counter with (256 – 125 = 131) as the initial starting count. It counts upwards 125 times every 8 usec until it overflows at 255 – this causes an interrupt. The initial count (131) is reloaded as part of the interrupt function and the process repeats itself.

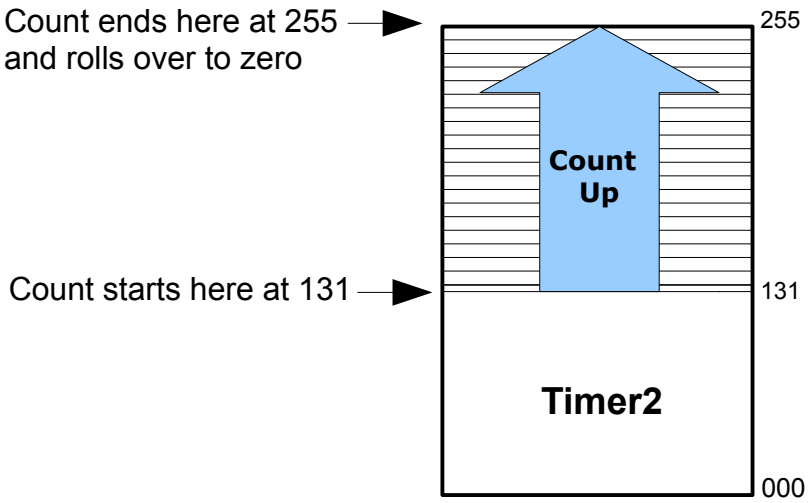


Figure 66: Timer2 Counts up, Interrupts when it overfows past 255

Fortunately, Swedish software whiz Sabastian Wallin has already worked out the Timer2 setup to get a 1.000 msec interrupt. His article “*Mastering Timer Interrupts on the Arduino*” can be found here:

http://popdevelop.com/2010/04/mastering-timer-interrupts-on-the-arduino/

In Sebastian's example, the interrupt function blinked an LED; we'll replace that with a D/A and Display SPI update. Here's Sebastian's code to set up a Timer2 interrupt.

```
 // First disable the timer overflow interrupt while we're configuring
TIMSK2 &= ~(1<<TOIE2);

// Configure timer2 in normal mode (pure counting, no PWM etc.)
TCCR2A &= ~((1<<WGM21) | (1<<WGM20));
TCCR2B &= ~(1<<WGM22);

// Select clock source: internal I/O clock
ASSR &= ~(1<<AS2);

// Disable Compare Match A interrupt enable (only want overflow)
TIMSK2 &= ~(1<<OCIE2A);

// Now configure the prescaler to CPU clock divided by 128
TCCR2B |= (1<<CS22)  | (1<<CS20);     // Set bits
TCCR2B &= ~(1<<CS21);                 // Clear bit

// We need to calculate a proper value to load the timer counter.
// The following loads the value 131 into the Timer 2 counter register
// The math behind this is:
// (CPU frequency) / (prescaler value) = 125000 Hz = 8us.
// (desired period) / 8us = 125.
// MAX(uint8) + 1 - 125 = 131;
//
// Save value globally for later reload in ISR  /
tcnt2 = 131;

// Finally load end enable the timer
TCNT2 = tcnt2;
TIMSK2 |= (1<<TOIE2);
```

Here is Sebastian's Interrupt Service Routine code which simply blinks an LED at 1.0 msec rate (observable as a square wave using an oscilloscope).

```
//  Install the Interrupt Service Routine (ISR) for Timer2 overflow.
// This is normally done by writing the address of the ISR in the
//  interrupt vector table but conveniently done by using ISR()
ISR(TIMER2_OVF_vect) {
    TCNT2 = tcnt2;                          // Reload the timer
    digitalWrite(2, toggle == 0 ? HIGH : LOW);      // toggle a digital pin
    toggle = ~toggle;
}
```

Remove these two statements and replace with code that will write the next waveform sample to the D/A and update the 7-segment numeric display.

A 5kΩ pot connected to an Analog Input is used to set the heart rate. While there is one floating point operation to calculate the idle period needed to achieve the selected heart rate, most heart rate data is integer, specifically in “times 10” format. In this motif, 427 represents a heart rate of 42.7 beats per minute. Noting that there are 543 samples in our PQRST waveform table, BpmLow is 30.0 beats per minute and BpmHigh is 110.4 beats per minute.

```
unsigned int  NumSamples = sizeof(y_data) / 2;          // number of elements in y_data[ ] above (543)
unsigned int  BpmLow;                                   // lowest heart rate allowed (x10)
unsigned int  BpmHigh;                                  // highest heart rate allowed (x10)
// establish the heart rate range allowed
// BpmLow  = 300          (30 bpm x 10)
// BpmHigh = (60.0 / (NumSamples * 0.001)) * 10  = (60.0 / .543) * 10 = 1104  (110.49 bpm x 10)
BpmLow = 300;
BpmHigh = (60.0 / ((float)NumSamples * 0.001)) * 10;
```

Creating the Background Loop Function

The Loop Function is a set of statements that are executed over and over again. In the ECG Simulator, the Loop Function is used to read the analog pot and calculate the number of samples needed in the “quiescent period” of the waveform to achieve the desired heart rate. It also calculates the desired heart rate in “beats per minute times 10” format.

Read and Map the Potentiometer

In the Setup Function, we calculated the value of BpmLow as 300 (30.0) beats per minute. We also calculated the value of BpmHigh as 1104 (110.4) beats per minute. Therefore, the A/D value for Analog Input 1 (raw input range 0 .. 1023) should be scaled to the range 300 .. 1104.

```
// read from the heart rate pot (Analog Input 0)
Value = analogRead(0);


// map the Analog Input 0 range (0 .. 1023) to the Bpm range (300 .. 1104)
Bpm = map(Value, 0, 1023, BpmLow, BpmHigh);
```

Filter the Analog Heart Rate Value

The Arduino Analog Input is subject to a fair amount of jitter and noise which tends to make the 4-digit display jump a bit in the two lower significant digits. An easy way to filter and suppress this jitter is a simple moving average filter. This is simply a 32 word circular buffer with wrap-around. The buffer pointer is advanced to the oldest entry and the new sample is placed there. If advancing to the next array element would exceed the length of the buffer, it automatically “wraps around” to the zeroth (or first) array element and enters it there. When a filtered value is needed, we simply sum all our 32 values in the array and divide by 32 (shift right 5 does the trick).

The filtered heart rate value is written to a variable DisplayValue which is only read by the Timer2 Interrupt Routine.

```
// To lessen the jitter or bounce in the display's least significant digit,
// a moving average filter (32 values) will smooth it out.
BpmValues[Index++] = Bpm;                 // add latest sample to 32 element array
if (Index == 32) {                        // handle wrap-around
  Index = 0;
}
BpmAverage = 0;
for (int  i = 0;  i < 32; i++) {          // summation of all values in the array
  BpmAverage += BpmValues[i];
}
BpmAverage >>= 5;                        // Divide by 32 to get average


// now update the 4-digit display - format: XXX.X
// since update is a multi-byte transfer, disable interrupts until it's done
noInterrupts();
DisplayValue = BpmAverage;
interrupts();
```

Create the ECG Waveform Data Structure

The y_data structure created by the Python post-processing program shown below gives 543 samples.

```
// ************************************************************************************************
//   y_data[543] - digitized ecg waveform, sampled at 1.0 msec
//
//   Waveform is scaled for a 12-bit D/A converter (0 .. 4096)
//
//   A 60 beat/min ECG would require this waveform (543 samples) plus 457 samples
//   of the first y_data[0] value of 936.
//
// ************************************************************************************************
const short  y_data[] = {
939, 940, 941, 942, 944, 945, 946, 947, 951, 956,
962, 967, 973, 978, 983, 989, 994, 1000, 1005, 1015,
1024, 1034, 1043, 1053, 1062, 1075, 1087, 1100, 1112, 1121,
1126, 1131, 1136, 1141, 1146, 1151, 1156, 1164, 1172, 1179,
1187, 1194, 1202, 1209, 1216, 1222, 1229, 1235, 1241, 1248,
1254, 1260, 1264, 1268, 1271, 1275, 1279, 1283, 1287, 1286,
1284, 1281, 1279, 1276, 1274, 1271, 1268, 1266, 1263, 1261,
1258, 1256, 1253, 1251, 1246, 1242, 1237, 1232, 1227, 1222,
1218, 1215, 1211, 1207, 1203, 1199, 1195, 1191, 1184, 1178,
1171, 1165, 1159, 1152, 1146, 1141, 1136, 1130, 1125, 1120,
1115, 1110, 1103, 1096, 1088, 1080, 1073, 1065, 1057, 1049,
1040, 1030, 1021, 1012, 1004, 995, 987, 982, 978, 974,
970, 966, 963, 959, 955, 952, 949, 945, 942, 939,
938, 939, 940, 941, 943, 944, 945, 946, 946, 946,
946, 946, 946, 946, 946, 947, 950, 952, 954, 956,
958, 960, 962, 964, 965, 965, 965, 965, 965, 965,
963, 960, 957, 954, 951, 947, 944, 941, 938, 932,
926, 920, 913, 907, 901, 894, 885, 865, 820, 733,
606, 555, 507, 632, 697, 752, 807, 896, 977, 1023,
1069, 1127, 1237, 1347, 1457, 2085, 2246, 2474, 2549, 2595,
2641, 2695, 3083, 3135, 3187, 3217, 3315, 3403, 3492, 3581,
3804, 3847, 3890, 3798, 3443, 3453, 3297, 3053, 2819, 2810,
2225, 2258, 1892, 1734, 1625, 998, 903, 355, 376, 203,
30, 33, 61, 90, 119, 160, 238, 275, 292, 309,
325, 343, 371, 399, 429, 484, 542, 602, 652, 703,
758, 802, 838, 856, 875, 895, 917, 938, 967, 1016,
1035, 1041, 1047, 1054, 1060, 1066, 1066, 1064, 1061, 1058,
1056, 1053, 1051, 1048, 1046, 1043, 1041, 1038, 1035, 1033,
1030, 1028, 1025, 1022, 1019, 1017, 1014, 1011, 1008, 1006,
1003, 1001, 999, 998, 996, 994, 993, 991, 990, 988,
986, 985, 983, 981, 978, 976, 973, 971, 968, 966,
963, 963, 963, 963, 963, 963, 963, 963, 963, 963,
963, 963, 963, 963, 963, 963, 963, 963, 963, 963,
964, 965, 966, 967, 968, 969, 970, 971, 972, 974,
976, 978, 980, 983, 985, 987, 989, 991, 993, 995,
997, 999, 1002, 1006, 1011, 1015, 1019, 1023, 1028, 1032,
1036, 1040, 1045, 1050, 1055, 1059, 1064, 1069, 1076, 1082,
1088, 1095, 1101, 1107, 1114, 1120, 1126, 1132, 1141, 1149,
1158, 1166, 1173, 1178, 1183, 1188, 1193, 1198, 1203, 1208,
1214, 1221, 1227, 1233, 1240, 1246, 1250, 1254, 1259, 1263,
1269, 1278, 1286, 1294, 1303, 1309, 1315, 1322, 1328, 1334,
1341, 1343, 1345, 1347, 1349, 1351, 1353, 1355, 1357, 1359,
1359, 1359, 1359, 1359, 1358, 1356, 1354, 1352, 1350, 1347,
1345, 1343, 1341, 1339, 1336, 1334, 1332, 1329, 1327, 1324,
1322, 1320, 1317, 1315, 1312, 1307, 1301, 1294, 1288, 1281,
1275, 1270, 1265, 1260, 1256, 1251, 1246, 1240, 1233, 1227,
1221, 1214, 1208, 1201, 1194, 1186, 1178, 1170, 1162, 1154,
1148, 1144, 1140, 1136, 1131, 1127, 1123, 1118, 1114, 1107,
1099, 1090, 1082, 1074, 1069, 1064, 1058, 1053, 1048, 1043,
1038, 1034, 1029, 1025, 1021, 1017, 1013, 1009, 1005, 1001,
997, 994, 990, 991, 992, 994, 996, 997, 999, 998,
997, 996, 995, 994, 993, 991, 990, 989, 989, 989,
989, 989, 989, 989, 988, 986, 984, 983, 981, 980,
982, 984, 986, 988, 990, 993, 995, 997, 999, 1002,
1005, 1008, 1012};
```

Determine the Number of Samples in the ECG Waveform

There's no need to count the number of samples in the rather hefty initializer above, we can use the C Language sizeof( ) function to do that. Since each sample is a 16-bit integer and sizeof( ) returns bytes, a divide-by-2 gives the actual number of samples in the waveform.

```
// global variables used by the program
unsigned int  NumSamples = sizeof(y_data) / 2;          // number of elements in y_data[] above (543)
```

Calculate the Number of Samples in the Quiescent Period

The following equation converts the selected heart rate (30 bpm to 110.4 bpm) to the number of quiescent samples required. In this situation, zero quiescent samples gives the fastest heart rate of 110.4 bpm; 1457 quiescent period samples gives the slowest heart rate of 30.0 bpm.

$$IdlePeriod = \frac{60000.0}{BeatsPerMinute} - NumSamples$$

```
// given the pot value (beats per minute) read in, calculate the IdlePeriod (msec)
// this value is used by the Timer2 1.0 msec interrupt service routine
BeatsPerMinute = (float)Bpm / 10.0;
noInterrupts();
IdlePeriod = (unsigned int)((float)60000.0 / BeatsPerMinute) – (float)NumSamples;
interrupts();
```

The Interrupt Function

When Timer2 overflows, it asserts a Timer2 interrupt. A special function called an “interrupt service routine” can be defined that will suspend the currently executing Arduino program and start the interrupt function immediately. Well, not exactly immediately, but within a few microseconds (this is called interrupt latency). The interrupt function will do the following things:

1. Read the next sample from the waveform data array.
2. Send the waveform sample to the D/A converter via the SPI interface.
3. Every 50th entry to the interrupt function, send the heart rate to the display via SPI.

When the interrupt function has completed the work listed above, it exits and resumes execution from the place it was when the Timer2 interrupt occurred. As you might imagine, the Timer2 always interrupts whatever is going on in the Arduino “loop” function.

The interrupt routine operates in three modes (setup, QRS waveform generation, and quiescent period waveform generation). Therefore, it's best to design the interrupt function as a “state machine”.

State machines can be emulated in software with a “switch” statement, as shown below.

```
ISR(TIMER2_OVF_vect) {

// Reload the timer
TCNT2 = tcnt2;

// state machine
switch (State) {

    case INIT:
        // zero the QRS and IDLE counters
        QRSCount = 0;
        IdleCount = 0;
        DisplayCount = 0;

        // set next state to QRS
        State = QRS;
    break;

    case QRS:

      // output the next sample in the QRS waveform to the D/A converter
      DTOA_Send(y_data[QRSCount]);

      // advance sample counter and check for end
      QRSCount++;
      if (QRSCount >= NumSamples) {
          // start IDLE period and output first sample to DTOA
          QRSCount = 0;
          DTOA_Send(y_data[0]);
          State = IDLE;
       }
    break;

    case IDLE:
        // since D/A converter will hold the previous value written, all we have
        // to do is determine how long the IDLE period should be.

        // advance idle counter and check for end
        IdleCount++;

       // the IdlePeriod is calculated in the main loop (from a pot)
       if (IdleCount >= IdlePeriod) {
           IdleCount = 0;
           State = QRS;
       }
    break;

  default:
  break;
}

 // output to the 7-segment display every 50 msec
DisplayCount++;
if (DisplayCount >= 50) {
    DisplayCount = 0;
    Display7Seg_Send(DisplayValue);
}
}
```

This is how we reload the Timer2 counter with 131 over and over again

The states are INIT, QRS and IDLE)

Init state is executed only once.

Send the QRS sample to the D/A

Detect last QRS sample and output first QRS sample to D/A. This value will hold for the duration of the Idle period.

Count out the Idle period and then switch to QRS state.

Every 50th entry to Timer2 Interrupt function, update the 4-digit display.

Figure 67: Timer2 Interrupt Function

DTOA_Send Function

The Timer2 Interrupt Function sends a new sample to the D/A converter every 1.0 millisecond. SPI operations interfacing to the D/A were grouped into a separate function. The data sheet for the Microchip MCP4921 converter requires that 16 bits be sent to the D/A via the SPI interface. This includes 4 configuration bits and the 12 waveform sample data bits as shown in Figure 68 below.

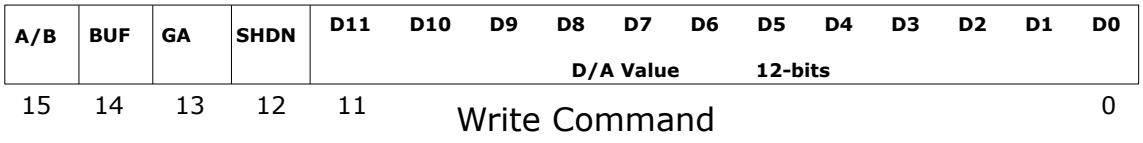


Figure 68: D/A Converter Write Command - SPI

Bits 12 – 15 of the D/A command are configuration bits. The table below shows the proper setting for these bits. Basically, 0011 should always be written to bits 15 – 12.

bit 15	A/B:	DACA or DACB Select bit
		1 = Write to DACB
		0 = Write to DACA **(There is only one D/A converter on the MCP4921 – Choose 0 for A/B)**
bit 14	BUF:	VREF Input Buffer Control bit
		1 = Buffered
		0 = Unbuffered **(Unbuffered mode allows 0 to 5 volt operation – Choose 0 for BUF)**
bit 13	GA:	Output Gain Select bit
		1 = 1x (VOUT = VREF * D/4096) **(1x gain gives the output range 0 to 5 volts – Choose 1 for GA)**
		0 = 2x (VOUT = 2 * VREF * D/4096)
bit 12	SHDN:	Output Power Down Control bit
		1 = Output Power Down Control bit **(D/A will not output anything unless SHDN is 1 – Choose 1 for SHDN)**
		0 = Output buffer disabled, Output is high impedance

Table 1: Microchip MCP4921 D/A Converter Configuration Bits

Based on the suggested configuration bits shown in Table 1, the data sent to the D/A converter via the SPI interface is shown in Figure 69 below. Since the Arduino has an 8-bit SPI interface, the 16 bits of D/A data must be sent in two 8-bit packets (bits 15 – 8 first, followed by bits 7 – 0).

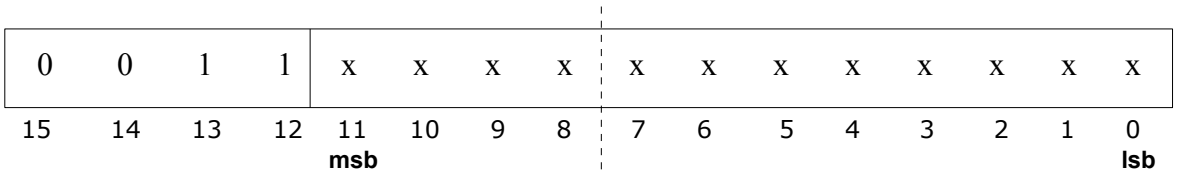


Figure 69: D/A Converter Write Command - SPI

Note in the D/A driver shown below that the chip is selected by setting Digital Output 10 low to select the chip and at the end setting it high to deselect the D/A chip. The author measured the execution time of this function with an oscilloscope and found it to be about 63 microseconds.

```
void  DTOA_Send(unsigned short DtoAValue) {

 byte          Data = 0;
 // select the D/A chip (low)
 digitalWrite(10, 0);    // chip select low

 // send the high byte first 0011xxxx
 Data = highByte(DtoAValue);
 Data = 0b00001111 & Data;
 Data = 0b00110000 | Data;
 SPI.transfer(Data);

 // send the low byte next xxxxxxxx
 Data = lowByte(DtoAValue);
 SPI.transfer(Data);

 // all done, de-select the chip (this updates the D/A with the new value)
 digitalWrite(10, 1);    // chip select high
}
```

Figure 70: D/A Converter SPI Driver

Display7Seg_Send Function

The Sparkfun Serial 7 Segment Display utilizes a ATMega328 microprocessor to receive and analyze incoming SPI serial commands to illuminate the proper segments and decimal points within the 4 digit display. The device purchased for this project was a blue color, but you can purchase other colors such as yellow, red and green.

To illuminate the numbers, just send four bytes to the device. These bytes can be the decimal numbers 0 to 9 or the ASCII equivalents of 0x30 to 0x39 (the on-board microprocessor figures this out by context). So, for example, to send 1234 to the display, one could send this sequence:

```
SPI.transfer(1);
SPI.transfer(2);
SPI.transfer(3);
SPI.transfer(4);
```

To send a “blank” digit, send the hex code 0x78:

```
SPI.transfer(0x78);
```

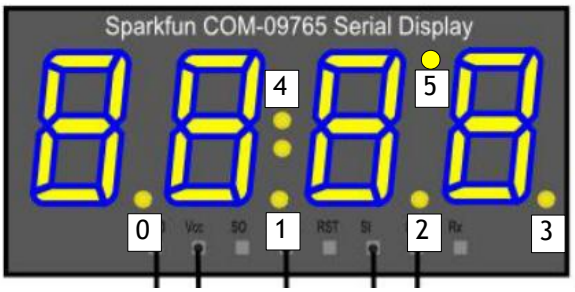
To reset the display, send the hex code 0x76. This will clear all digits and decimal points and set the cursor to the left-most digit.

```
SPI.transfer(0x76);
```

The brightness can be adjusted digitally but it requires a two-byte sequence. First send the hex code 0x7A (set brightness) followed by a byte value signifying the desired brightness from 0 to 255 (255 being the brightest setting).

```
SPI.transfer(0x7A);
SPI.transfer(255);
```

Illuminating the decimal points also requires a two-byte sequence. First send the hex code 0x77 (set decimal points) followed by a byte value signifying the decimal points to be illuminated. Figure 71 shows the relationship between the decimal points and the second byte.



Decimal Point Bit Field

0	0	5	4	3	2	1	0
7	6	5	4	3	2	1	0

Figure 71: Bit Field to Illuminate the Decimal Points

For example, to illuminate the decimal point between the third and fourth digits (labeled “2” in Figure 71 above), the bit field would be encoded as:

0 0 0 0 0 1 0 0 = 0x04

Thus one would send the following sequence to illuminate decimal point # 2:

```
SPI.transfer(0x77);
SPI.transfer(0x04);
```

Illuminating the colon turns on both dots of the colon, you cannot address them individually. Figure 72 below shows the completed code for the Display7Seg_Send function. There is a bit of logic included to suppress leading zeros.

```
void  Display7Seg_Send(unsigned int  HeartRate) {
 uint8_t       digit1, digit2, digit3, digit4;
 unsigned int  value;

 // convert to four digits (set leading zeros to blanks; 0x78 is the blank character)
 value = HeartRate;
 digit1 = value / 1000;
 value -= digit1 * 1000;
 if (digit1 == 0) digit1 = 0x78;

 digit2 = value / 100;
 value -= digit2 * 100;
 if ((digit1 == 0x78) && (digit2 == 0)) digit2 = 0x78;

 digit3 = value / 10;
 value -= digit3 * 10;
 if ((digit1 == 0x78) && (digit2 == 0x78) && (digit3 == 0)) digit3 = 0x78;

 digit4 = value;

 digitalWrite(9, LOW);    // select the Sparkfun 7-seg display
 SPI.transfer(0x76);      // reset display
 SPI.transfer(0x7A);      // brightness command
 SPI.transfer(0x00);      // 0 = bright,  255 = dim
 SPI.transfer(digit1);    // Thousands Digit
 SPI.transfer(digit2);    // Hundreds Digit
 SPI.transfer(digit3);    // Tens Digit
 SPI.transfer(digit4);    // Ones Digit
 SPI.transfer(0x77);      // set decimal points command
 SPI.transfer(0x04);      // turn on dec pt between digits 3 and 4
 digitalWrite(9, HIGH);   // release Sparkfun 7-seg display
}
```

Figure 72: 4 Digit Numeric Display Driver

ECG SIMULATOR ARDUINO SKETCH

The following is a listing of the complete Arduino sketch for the Adafruit Menta ECG Simulator. You can access this sketch on the “Lynchzilla” GitHub account.

```
// ************************************************************************************************************************
//                              ECG SIMULATOR
//
//  Purpose: simulate the normal sinus rhythm ECG signal (3-leads RA,LA,RL)
//
//  Background:
//
//  In normal electrocardiography (ECG or EKG if you're German), three leads make up the
//  Einthoven's triangle. Two leads are taped to the right and left side of the chest above
//  the heart (RA = right arm, LA = left arm) and one lead is taped to the lower chest, typically
//  on the right hip (RL = right leg).
//
//  It's important to know that these ECG signals are millivolts in amplitude. This can be achieved by
//  feeding the D/A converter through a voltage divider to get to the millivolt levels.
//
//
//  The ECG signal:
//
//  I found a suitable ECG waveform from the internet. Here is how I converted a picture from my
//  monitor screen to a C language array of A/D values, each spaced 1.00 msec apart.
//
//      A. Screen shot of waveform using the free screen capture program MWSNAP
//              http://www.mirekw.com/winfreeware/mwsnap.html
//
//      B. Digitize the jpeg waveform using the free digitizing program ENGAUGE
//              http://digitizer.sourceforge.net/
//
//      C: I wrote a Python program to convert the rather irregular samples from ENGAUGE
//         to an array of values spaced 1.0 milliseconds apart using linear interpolation.
//         Then I created a text file where these data points were part of a C language array
//         construct; that is, the data points are C initializers.
//
//      D: Cut-and-paste from the text file the C data array with initializers into the
//         Arduino sketch below.
//
```

```
//
//  Arduino Resources:
//
//    Digital Output #  9  - chip select the 7-segment display SPI port (low to select)
//    Digital Output # 10 - chip select for D/A converter  (low to select)
//    Digital Output # 11 - SDI data to the D/A converter  (SPI interface)
//    Digital Output # 13 - SCK clock to the D/A converter (SPI interface)
//
//    Analog Input   # 0  - center wiper pin of 5k ohm pot (heart rate adjust)
//
//    I followed the Timer2 setup as outlined by Sebastian Wallin
//    http://popdevelop.com/2010/04/mastering-timer-interrupts-on-the-arduino/
//
//    I set up the SPI interface according to the excellent instructions of Australian John Boxall,
//    whose wonderful website has many excellent Arduino tutorials:
//    http://tronixstuff.wordpress.com/
//
//  Programmer:  James P Lynch
//                       lynch007@gmail.com
//
// ********************************************************************************************************************
#include "SPI.h"          // supports the SPI interface to the D/A converter and 7-segment display
#include <Wire.h>         // need the Wire library

// various constants used by the waveform generator
#define INIT          0
#define IDLE          1
#define QRS           2
#define FOUR          4
#define THREE         3
#define TWO           2
#define ONE           1

// ********************************************************************************************************************
//   y_data[543] - digitized ecg waveform, sampled at 1.0 msec
//
//   Waveform is scaled for a 12-bit D/A converter (0 .. 4096)
//
//   A 60 beat/min ECG would require this waveform (543 samples) plus 457 samples
//   of the first y_data[0] value of 936.
// ********************************************************************************************************************
const short  y_data[] = {
939, 940, 941, 942, 944, 945, 946, 947, 951, 956,
962, 967, 973, 978, 983, 989, 994, 1000, 1005, 1015,
1024, 1034, 1043, 1053, 1062, 1075, 1087, 1100, 1112, 1121,
1126, 1131, 1136, 1141, 1146, 1151, 1156, 1164, 1172, 1179,
1187, 1194, 1202, 1209, 1216, 1222, 1229, 1235, 1241, 1248,
1254, 1260, 1264, 1268, 1271, 1275, 1279, 1283, 1287, 1286,
```

```
1284, 1281, 1279, 1276, 1274, 1271, 1268, 1266, 1263, 1261,
1258, 1256, 1253, 1251, 1246, 1242, 1237, 1232, 1227, 1222,
1218, 1215, 1211, 1207, 1203, 1199, 1195, 1191, 1184, 1178,
1171, 1165, 1159, 1152, 1146, 1141, 1136, 1130, 1125, 1120,
1115, 1110, 1103, 1096, 1088, 1080, 1073, 1065, 1057, 1049,
1040, 1030, 1021, 1012, 1004, 995, 987, 982, 978, 974,
970, 966, 963, 959, 955, 952, 949, 945, 942, 939,
938, 939, 940, 941, 943, 944, 945, 946, 946, 946,
946, 946, 946, 946, 946, 947, 950, 952, 954, 956,
958, 960, 962, 964, 965, 965, 965, 965, 965, 965,
963, 960, 957, 954, 951, 947, 944, 941, 938, 932,
926, 920, 913, 907, 901, 894, 885, 865, 820, 733,
606, 555, 507, 632, 697, 752, 807, 896, 977, 1023,
1069, 1127, 1237, 1347, 1457, 2085, 2246, 2474, 2549, 2595,
2641, 2695, 3083, 3135, 3187, 3217, 3315, 3403, 3492, 3581,
3804, 3847, 3890, 3798, 3443, 3453, 3297, 3053, 2819, 2810,
2225, 2258, 1892, 1734, 1625, 998, 903, 355, 376, 203,
30, 33, 61, 90, 119, 160, 238, 275, 292, 309,
325, 343, 371, 399, 429, 484, 542, 602, 652, 703,
758, 802, 838, 856, 875, 895, 917, 938, 967, 1016,
1035, 1041, 1047, 1054, 1060, 1066, 1066, 1064, 1061, 1058,
1056, 1053, 1051, 1048, 1046, 1043, 1041, 1038, 1035, 1033,
1030, 1028, 1025, 1022, 1019, 1017, 1014, 1011, 1008, 1006,
1003, 1001, 999, 998, 996, 994, 993, 991, 990, 988,
986, 985, 983, 981, 978, 976, 973, 971, 968, 966,
963, 963, 963, 963, 963, 963, 963, 963, 963, 963,
963, 963, 963, 963, 963, 963, 963, 963, 963, 963,
964, 965, 966, 967, 968, 969, 970, 971, 972, 974,
976, 978, 980, 983, 985, 987, 989, 991, 993, 995,
997, 999, 1002, 1006, 1011, 1015, 1019, 1023, 1028, 1032,
1036, 1040, 1045, 1050, 1055, 1059, 1064, 1069, 1076, 1082,
1088, 1095, 1101, 1107, 1114, 1120, 1126, 1132, 1141, 1149,
1158, 1166, 1173, 1178, 1183, 1188, 1193, 1198, 1203, 1208,
1214, 1221, 1227, 1233, 1240, 1246, 1250, 1254, 1259, 1263,
1269, 1278, 1286, 1294, 1303, 1309, 1315, 1322, 1328, 1334,
1341, 1343, 1345, 1347, 1349, 1351, 1353, 1355, 1357, 1359,
1359, 1359, 1359, 1359, 1358, 1356, 1354, 1352, 1350, 1347,
1345, 1343, 1341, 1339, 1336, 1334, 1332, 1329, 1327, 1324,
1322, 1320, 1317, 1315, 1312, 1307, 1301, 1294, 1288, 1281,
1275, 1270, 1265, 1260, 1256, 1251, 1246, 1240, 1233, 1227,
1221, 1214, 1208, 1201, 1194, 1186, 1178, 1170, 1162, 1154,
1148, 1144, 1140, 1136, 1131, 1127, 1123, 1118, 1114, 1107,
1099, 1090, 1082, 1074, 1069, 1064, 1058, 1053, 1048, 1043,
1038, 1034, 1029, 1025, 1021, 1017, 1013, 1009, 1005, 1001,
997, 994, 990, 991, 992, 994, 996, 997, 999, 998,
997, 996, 995, 994, 993, 991, 990, 989, 989, 989,
989, 989, 989, 989, 988, 986, 984, 983, 981, 980,
982, 984, 986, 988, 990, 993, 995, 997, 999, 1002,
1005, 1008, 1012};
```

```
// global variables used by the program
unsigned int     NumSamples = sizeof(y_data) / 2;          // number of elements in y_data[] above
unsigned int     QRSCount = 0;                             // running QRS period msec count
unsigned int     IdleCount = 0;                            // running Idle period msec count
unsigned long    IdlePeriod = 0;                           // idle period is adjusted by pot to set heart rate
unsigned int     State = INIT;                             // states are INIT, QRS, and IDLE
unsigned int     DisplayCount = 0;                         // counts 50 msec to update the 7-segment display
unsigned int     tcnt2;                                    // Timer2 reload value, globally available
float            BeatsPerMinute;                           // floating point representation of the heart rate
unsigned int     Bpm;                                      // integer version of heart rate (times 10)
unsigned int     BpmLow;                                   // lowest heart rate allowed (x10)
unsigned int     BpmHigh;                                  // highest heart rate allowed (x10)
int              Value;                                    // place holder for analog input 0
unsigned long BpmValues[32] = {0, 0, 0, 0, 0, 0, 0, 0,     // holds 32 last analog pot readings
                               0, 0, 0, 0, 0, 0, 0, 0,     // for use in filtering out display jitter
                               0, 0, 0, 0, 0, 0, 0, 0,     // for use in filtering out display jitter
                               0, 0, 0, 0, 0, 0, 0, 0};    // for use in filtering out display jitter
unsigned long BpmAverage = 0;                              // used in a simple averaging filter
unsigned char Index = 0;                                   // used in a simple averaging filter
unsigned int  DisplayValue = 0;                            // filtered Beats Per Minute sent to display

void setup()  {
  // Configure the output ports (1 msec intrerrupt indicator and D/A SPI support)
 pinMode(9, OUTPUT);              // 7-segment display chip select   (low to select chip)
 pinMode(10, OUTPUT);             // D/A converter chip select       (low to select chip)
 pinMode(11, OUTPUT);             // SDI data
 pinMode(13, OUTPUT);             // SCK clock

 // initial state of SPI interface
 SPI.begin();                                 // wake up the SPI bus.
 SPI.setDataMode(0);                          // mode: CPHA=0, data captured on clock's rising edge (low→high)
 SPI.setClockDivider(SPI_CLOCK_DIV64);  // system clock / 64
 SPI.setBitOrder(MSBFIRST);                   // bit 7 clocks out first

 // establish the heart rate range allowed
 // BpmLow  = 300 (30 bpm x 10)
 // BpmHigh = (60.0 / (NumSamples * 0.001)) * 10  = (60.0 / .543) * 10 = 1104  (110.49 x 10)
 BpmLow = 300;
 BpmHigh = (60.0 / ((float)NumSamples * 0.001)) * 10;

 // First disable the timer overflow interrupt while we're configuring
 TIMSK2 &= ~(1<<TOIE2);

 // Configure timer2 in normal mode (pure counting, no PWM etc.)
 TCCR2A &= ~((1<<WGM21) | (1<<WGM20));
 TCCR2B &= ~(1<<WGM22);

 // Select clock source: internal I/O clock
 ASSR &= ~(1<<AS2);
```

```
 // Disable Compare Match A interrupt enable (only want overflow)
 TIMSK2 &= ~(1<<OCIE2A);

 // Now configure the prescaler to CPU clock divided by 128
 TCCR2B |= (1<<CS22)  | (1<<CS20);            // Set bits
 TCCR2B &= ~(1<<CS21);                        // Clear bit

 // We need to calculate a proper value to load the timer counter.
 // The following loads the value 131 into the Timer 2 counter register
 // The math behind this is:
 // (CPU frequency) / (prescaler value) = 125000 Hz = 8us.
 // (desired period) / 8us = 125.
 // MAX(uint8) + 1 - 125 = 131;
 //
 // Save value globally for later reload in ISR  /
 tcnt2 = 131;
 // Finally load end enable the timer
 TCNT2 = tcnt2;
 TIMSK2 |= (1<<TOIE2);
}

void loop() {

 // read from the heart rate pot (Analog Input 0)
 Value = analogRead(0);

 // map the Analog Input 0 range (0 .. 1023) to the Bpm range (300 .. 1104)
 Bpm = map(Value, 0, 1023, BpmLow, BpmHigh);

 // To lessen the jitter or bounce in the display's least significant digit,
 // a moving average filter (32 values) will smooth it out.
 BpmValues[Index++] = Bpm;                    // add latest sample to 32 element array
 if (Index == 32) {                           // handle wrap-around
   Index = 0;
 }
 BpmAverage = 0;
 for (int  i = 0;  i < 32; i++) {             // summation of all values in the array
   BpmAverage += BpmValues[i];
 }
 BpmAverage >>= 5;                           // divide by 32 to get average

 // now update the 4-digit display - format: XXX.X
 // since update is a multi-byte transfer, disable interrupts until it's done
 noInterrupts();
 DisplayValue = BpmAverage;
 interrupts();

 // given the pot value (beats per minute) read in, calculate the IdlePeriod (msec)
 // this value is used by the Timer2 1.0 msec interrupt service routine
```

```
 BeatsPerMinute = (float)Bpm / 10.0;
 noInterrupts();
 IdlePeriod = (unsigned int)((float)60000.0 / BeatsPerMinute) - (float)NumSamples;
 interrupts();

 delay(20);
}

// ********************************************************************************
//              Timer2 Interrupt Service Routine
//
// Interrupt Service Routine (ISR) for Timer2 overflow at 1.000 msec.
//
//
// The Timer2 interrupt function is used to send the 16-bit waveform point
// to the Microchip MCP4921 D/A converter using the SPI interface.
//
// The Timer2 interrupt function is also used to send the current heart rate
// as read from the potentiometer every 50 Timer2 interrupts to the 7-segment display.
//
// The pot is read and the heart rate is calculated in the background loop.
// By running both SPI peripherals at interrupt level, we "serialize" them and avoid
// corruption by one SPI transmission being interrupted by the other.
//
// A state machime is implemented to accomplish this. It's states are:
//
//      INIT -      basically clears the counters and sets the state to QRS.
//
//      QRS  -    outputs the next ECG waveform data point every 1.0 msec
//                there are 543 of these QRS complex data points.
//
//      IDLE -    variable period after the QRS part.
//                D/A holds first ECG value (936) for all of the IDLE period.
//                Idle period varies to allow adjustment of the basic heart rate;
//                a value of zero msec for the idle period gives 110.4 beats per min
//                while the maximum idle period of 457 msec gives 30.0 bpm.
//
//                Note that the IDLE period is calculated in the main background
//                loop by reading a pot and converting its range to one suitable
//                for the background period. The interrupt routine reads this
//                value to determine when to stop the IDLE period.
//
// The transmission of the next data point to the D/A converter via SPI takes
// about 63 microseconds (that includes two SPI byte transmissions).
//
// The transmission of the heart rate digits to the Sparkfun 7-segment display
// takes about 350 usec (it is only transmitted every 50 Timer2 interrupts)
//
// ********************************************************************************
```

```
ISR(TIMER2_OVF_vect) {

 // Reload the timer
 TCNT2 = tcnt2;

 // state machine
 switch (State) {

  case INIT:

   // zero the QRS and IDLE counters
   QRSCount = 0;
   IdleCount = 0;
   DisplayCount = 0;

   // set next state to QRS
   State = QRS;
  break;

  case QRS:

   // output the next sample in the QRS waveform to the D/A converter
   DTOA_Send(y_data[QRSCount]);

   // advance sample counter and check for end
   QRSCount++;
   if (QRSCount >= NumSamples) {
    // start IDLE period and output first sample to DTOA
    QRSCount = 0;
    DTOA_Send(y_data[0]);
    State = IDLE;
   }
  break;

  case IDLE:

   // since D/A converter will hold the previous value written, all we have
   // to do is determine how long the IDLE period should be.

   // advance idle counter and check for end
   IdleCount++;

   // the IdlePeriod is calculated in the main loop (from a pot)
   if (IdleCount >= IdlePeriod) {
    IdleCount = 0;
    State = QRS;
   }
  break;
```

```
   default:
   break;
 }

 // output to the 7-segment display every 50 msec
 DisplayCount++;
 if (DisplayCount >= 50) {
   DisplayCount = 0;
   Display7Seg_Send(DisplayValue);
 }
}

// ***********************************************************************************************
//                   void  DTOA_Send(unsigned short)
//
//  Purpose: send 12-bit D/A value to Microchip MCP4921 D/A converter ( 0 .. 4096 )
//
//  Input:    DtoAValue - 12-bit D/A value ( 0 .. 4096 )
//
//          The DtoAValue is prepended with the A/B, BUF, GA, and SHDN bits before transmission.
//
//                                  WRITE COMMAND
//          |-------------|--------------|-------------|----------------|-----------------------------------------------------|
//          |  A/B      |   BUF       |  GA         | SHDN        | D11 D10 D09 D08 D07 D06 D05 D04 D03 D02 D01 D00  |
//          |             |              |             |               |                                                     |
//          |setting:  |setting:     |setting:    |Setting:     |          DtoAValue   (12 bits)                      |
//          |   0         |   0           |   1          |   1           |                                                     |
//          | DAC-A  |unbuffer    |   1x         |power-on    | ( 0 .. 4096  will output as 0 volts .. 5 volts )    |
//          |------------|---------------|--------------|----------------|-----------------------------------------------------|
//              15             14            13             12        11                                                 0
//  To D/A    <=============================================================================================
//
//          Note:  WriteCommand is clocked out with bit 15 first!
//
//  Returns:  nothing
//
//
//  I/O Resources:  Digital Pin 9  = chip select (low to select chip)
//                          Digital Pin 13 = SPI Clock
//                          Digital Pin 11 = SPI Data
//
//  Note: by grounding the LDAC* pin in the hardware hook-up, the SPI data will be clocked into the
//          D/A converter latches when the chip select rises at the end-of-transfer.
//
//       This routine takes 63 usec using an Adafruit Menta
// ***********************************************************************************************

void  DTOA_Send(unsigned short DtoAValue) {
 byte           Data = 0;
 // select the D/A chip (low)
```

```
 digitalWrite(10, 0);    // chip select low

 // send the high byte first 0011xxxx
 Data = highByte(DtoAValue);
 Data = 0b00001111 & Data;
 Data = 0b00110000 | Data;
 SPI.transfer(Data);

 // send the low byte next xxxxxxxx
 Data = lowByte(DtoAValue);
 SPI.transfer(Data);

 // all done, de-select the chip (this updates the D/A with the new value)
 digitalWrite(10, 1);    // chip select high
}

// ***************************************************************************************************
//                    void  Display7Seg_Send(char *)
//
//  Purpose: send 4 digits to SparkFun Serial 7-Segment Display (requires 4 SPI writes)
//
//  Input:    value -  unsigned int version of BeatsPerMinute
//
//  Returns:  nothing
//
//  I/O Resources:   Digital Pin 10 = chip select (low to select chip)
//                         Digital Pin 13 = SPI Clock
//                         Digital Pin 11 = SPI Data
//
//  Note:  this routine takes 350 usec using an Adafruit Menta
// ***************************************************************************************************
void  Display7Seg_Send(unsigned int  HeartRate) {
 uint8_t       digit1, digit2, digit3, digit4;
 unsigned int  value;

 // convert to four digits (set leading zeros to blanks; 0x78 is the blank character)
 value = HeartRate;
 digit1 = value / 1000;
 value -= digit1 * 1000;
 if (digit1 == 0) digit1 = 0x78;

 digit2 = value / 100;
 value -= digit2 * 100;
 if ((digit1 == 0x78) && (digit2 == 0)) digit2 = 0x78;

 digit3 = value / 10;
 value -= digit3 * 10;
 if ((digit1 == 0x78) && (digit2 == 0x78) && (digit3 == 0)) digit3 = 0x78;

 digit4 = value;
```

```
  digitalWrite(9, LOW);    // select the Sparkfun 7-seg display
  SPI.transfer(0x76);      // reset display
  SPI.transfer(0x7A);      // brightness command
  SPI.transfer(0x00);      // 0 = bright,  255 = dim
  SPI.transfer(digit1);    // Thousands Digit
  SPI.transfer(digit2);    // Hundreds Digit
  SPI.transfer(digit3);    // Tens Digit
  SPI.transfer(digit4);    // Ones Digit
  SPI.transfer(0x77);      // set decimal points command
  SPI.transfer(0x04);      // turn on dec pt between digits 3 and 4
  digitalWrite(9, HIGH);   // release Sparkfun 7-seg display
}
```

Test Results

To test the ECG simulator, the Texas Instruments ADS1293EVM evaluation board was connected to the Menta via the banana jacks and the supplied TI windows software was used to look at the received signal. The test setup is shown in Figure 73.

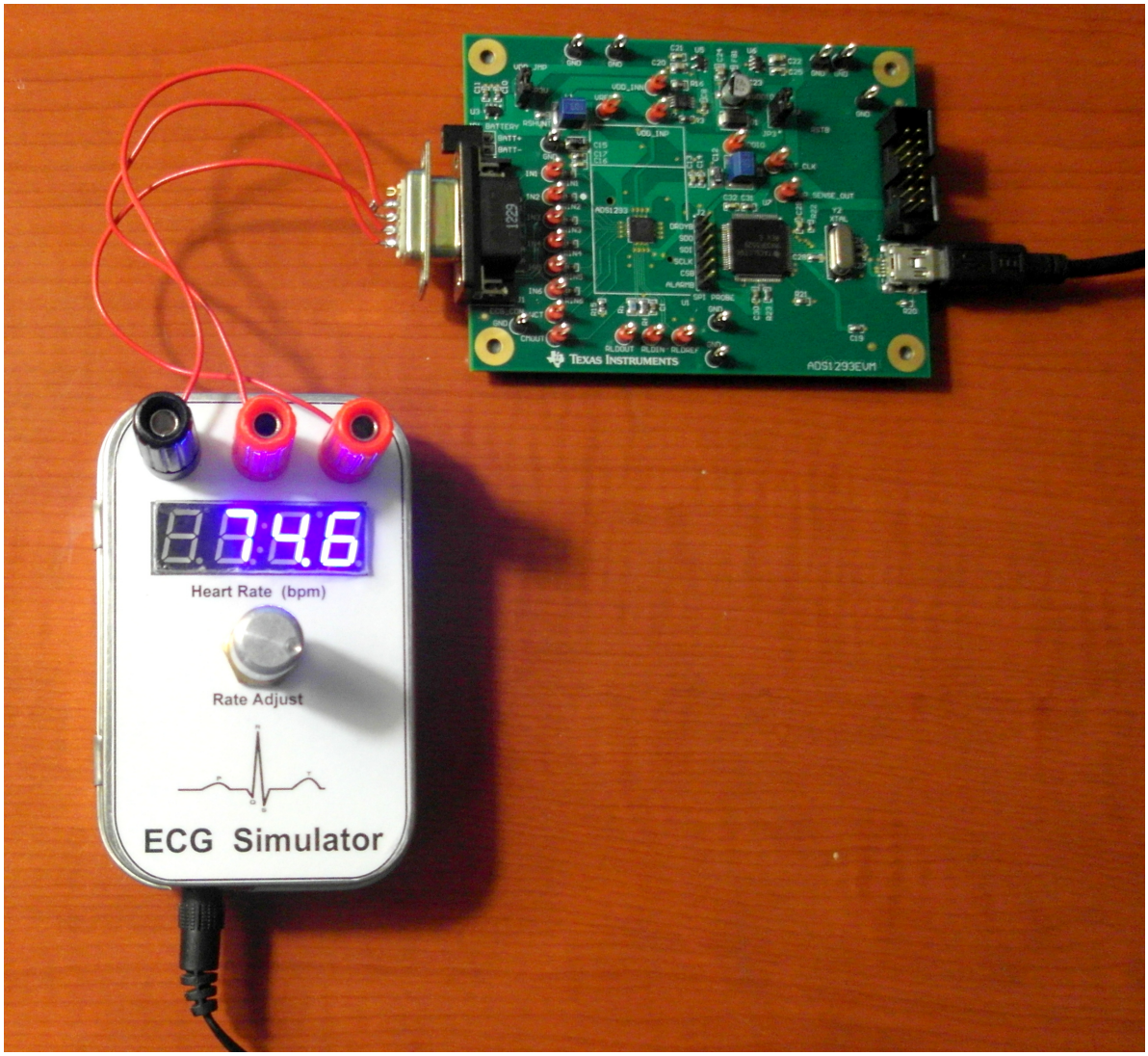


Figure 73: ECG Simulator connected to Texas Instruments ADS1239EVM Board

In the interest of honest reporting, heart monitors employ a lot of filtering to clean up the ECG signal. I set up the TI software digital filters to do the same thing. Figure 74 shows the Texas Instruments software displaying the ECG signal from the simulator.

Figure 75 shows the original ECG signal screen-captured from the Army document. As you can see, the generated ECG signal is a close match to the original. Note that the signal is just a couple of millivolts. With the Rate Adjustment pot, you can vary the rate from 30 bpm to 110 bpm.

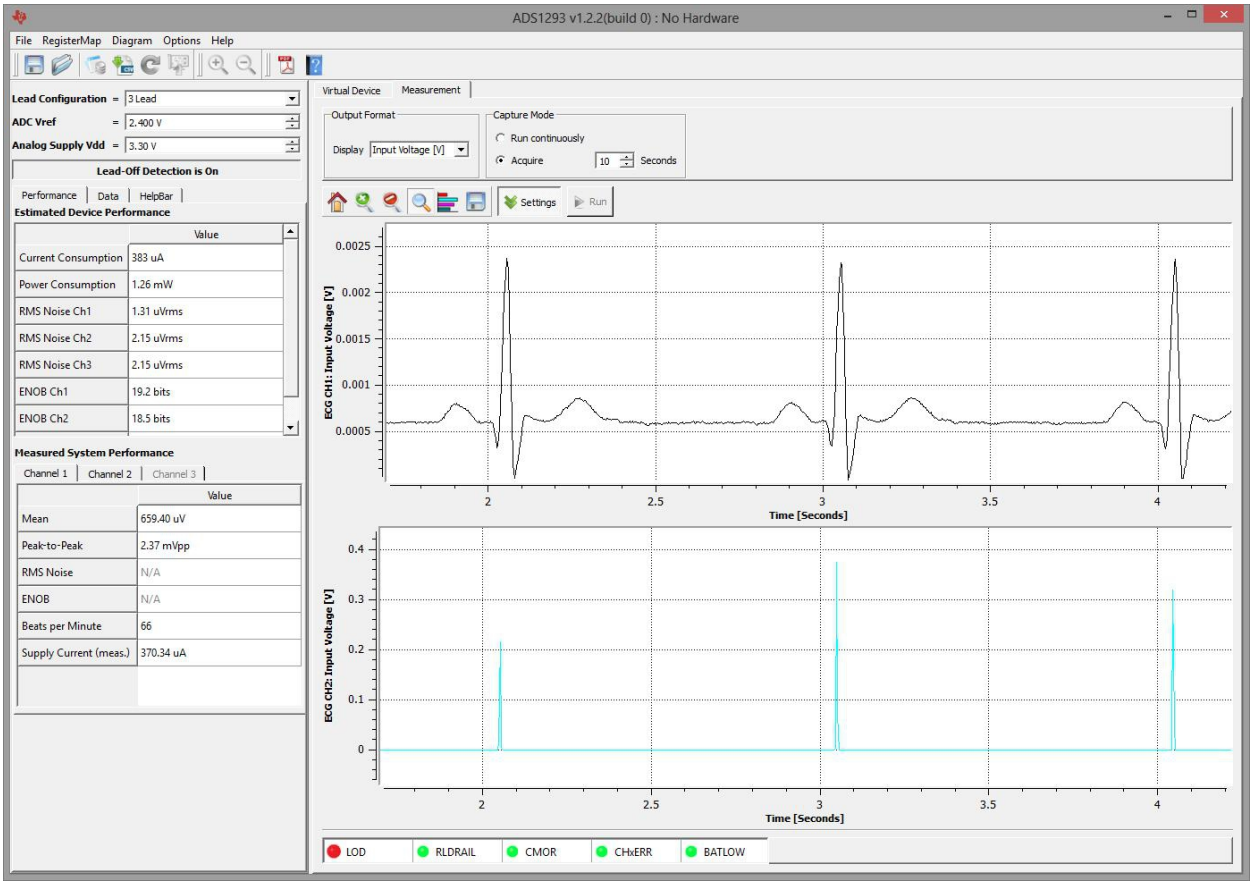


Figure 74: Received signal from the ECG Simulator (displayed by Texas Instruments Software)

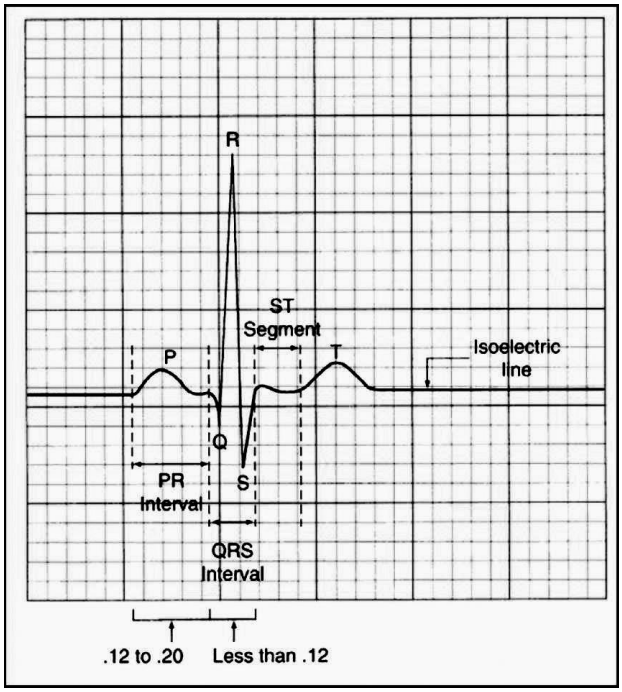


Figure 75: Original ECG waveform programmed into the Adafruit Menta

Conclusions

My goal with this project was to design and fabricate a 3-lead ECG simulator that would allow me to study the Texas Instruments ADS1293 ECG Front-end chip. Using the Adafruit Menta kit as a starting point, the simulator was built for just over $60 in parts. There was no intention to manufacture this device, it was essentially a “one-off” design.

The software effort was a bit more complicated since I decided to start from a page in an Army medical document on the Internet. The sample waveform was screen-captured using MWSnap, digitized by the Sourceforge Enguage tool, and then manipulated by a custom Python program to create a C language array structure with initializer that can be “pasted” into the Arduino sketch.

The Arduino sketch is modestly complicated, using a Timer2 interrupt to precisely time the waveform samples at one millisecond intervals and some SPI driver code to update the D/A converter and the 4-digit display.

There are two conclusions: the 8-bit Arduino micro-controller is a perfect fit for this application and the Adafruit Menta is a wonderful platform for building things.

It bears repeating that the strategies I outlined in this tutorial to generate the ECG waveform can be utilized to create any waveform that you might see in a book or on the Internet.

About the Author

Jim Lynch lives in Grand Island, New York and is a software developer for Control Techniques, a subsidiary of Emerson Electric. He develops embedded software for the company's industrial drives (high power motor controllers) which are sold all over the world.

Mr. Lynch has previously worked for Mennen Medical, Calspan Corporation, and the Boeing Company. He has a BSEE from Ohio University and a MSEE from State University of New York at Buffalo.

Jim is a single father and has two grown children and four grandchildren who now live in Florida and Nevada.

He has two brothers, one is a Viet Nam veteran in Hollywood, Florida and the other is the Bishop of St. Petersburg, also in Florida. Jim enjoys playing the guitar, woodworking, and going to the movies.

Lynch can be reached via e-mail at: lynch007@gmail.com

Appendix 1: MWSnap Screen Capture Utility

We write a lot of documentation as part of our work. Modern word processors allow rich use of graphics to enhance our presentations. One very popular graphic is the "screen shot", a recording of part or all of the computer display. Windows supports the "Print Screen" button that can record the entire screen, but this facility is crude and cumbersome to use. Sometimes we only want to capture just an area, a sub-window, or a pull-down menu.

There are lots of screen capture programs available, some free and some very expensive. I've tried many of them, but the screen capture utility **MWSnap** is outstanding in its ease of use. Best of all, it's free.

MWSnap was developed by Canadian software Engineer Mirek Wojtowicz who works for a transportation company called Turnpike Global Technologies in Oakville, Ontario.

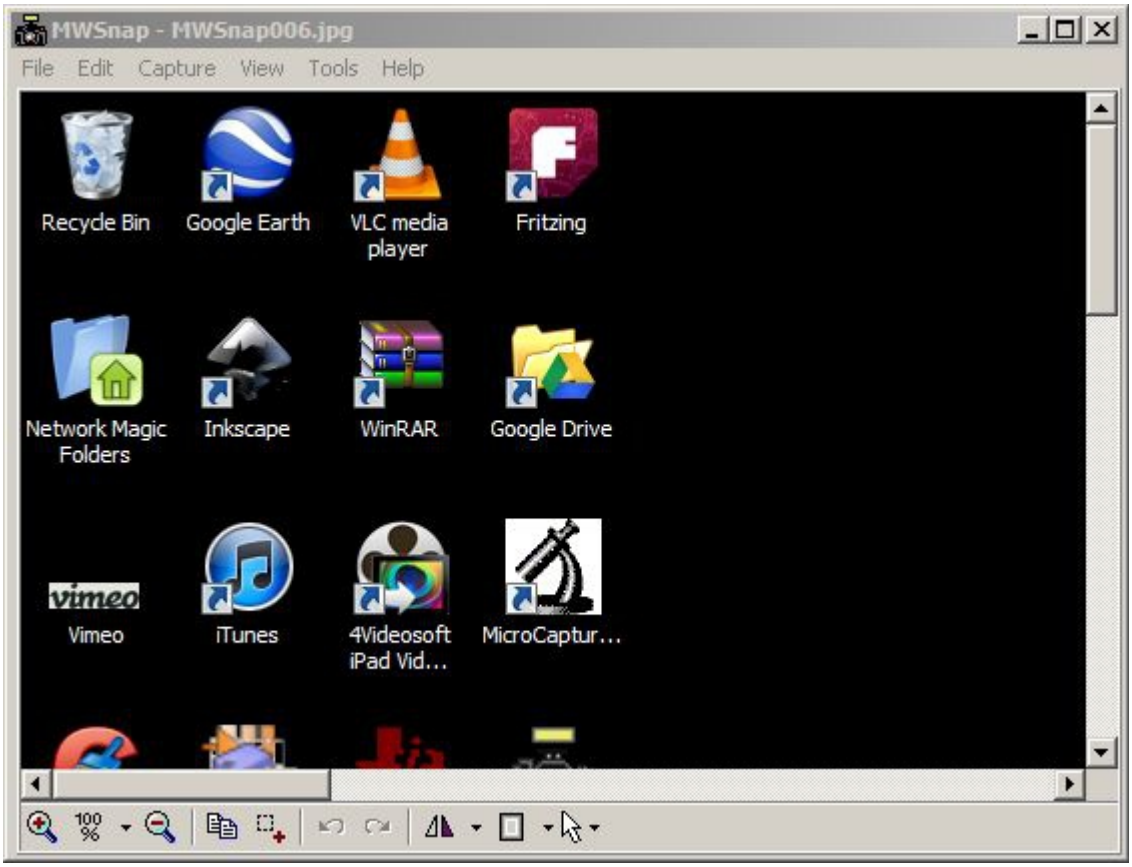


Figure 76: MWSnap Display

MWSnap Selected Features

- 5 snapping modes.
- Support for BMP, JPG, TIFF, PNG and GIF formats, with selected color depth and quality settings.
- System-wide hot keys.
- Clipboard copy/paste.
- Printing.
- Auto-saving, auto-printing.
- Auto-start with Windows.
- Minimizing to system tray.
- An auto-extending list of fixed sizes, perfect for snapping images for icons and glyphs.
- A zoom tool for magnifying selected parts of the screen.

- A ruler tool for measuring screen objects lengths.
- A color picker showing screen colors with separated RGB parts.
- Fast picture viewer.
- Adding frames and mouse pointer images.
- Multilevel configurable undo and redo.
- Multilingual versions.
- Configurable user interface.
- And more...

To use **MWSnap**, you just start it and minimize it. You can set up a couple of Function Keys to do an area capture or a window capture. If you strike the window capture function key, as you move the mouse cursor around the screen, **MWSnap** will draw a perimeter around the window it will capture, anything from the full screen to the smallest dialogue box. Clicking the left mouse button will capture the screen area you have chosen. **MWSnap** can be set up to automatically place the image as a jpeg file into a selected folder with a simple name such as "mwsnap001.jpg". The area capture mode is especially good at capturing pull- down menus.

Download and Install MWSnap

You can download the **MWSnap** program from the following link (it's just 643 kb):

http://www.mirekw.com/winfreeware/mwsnap.html

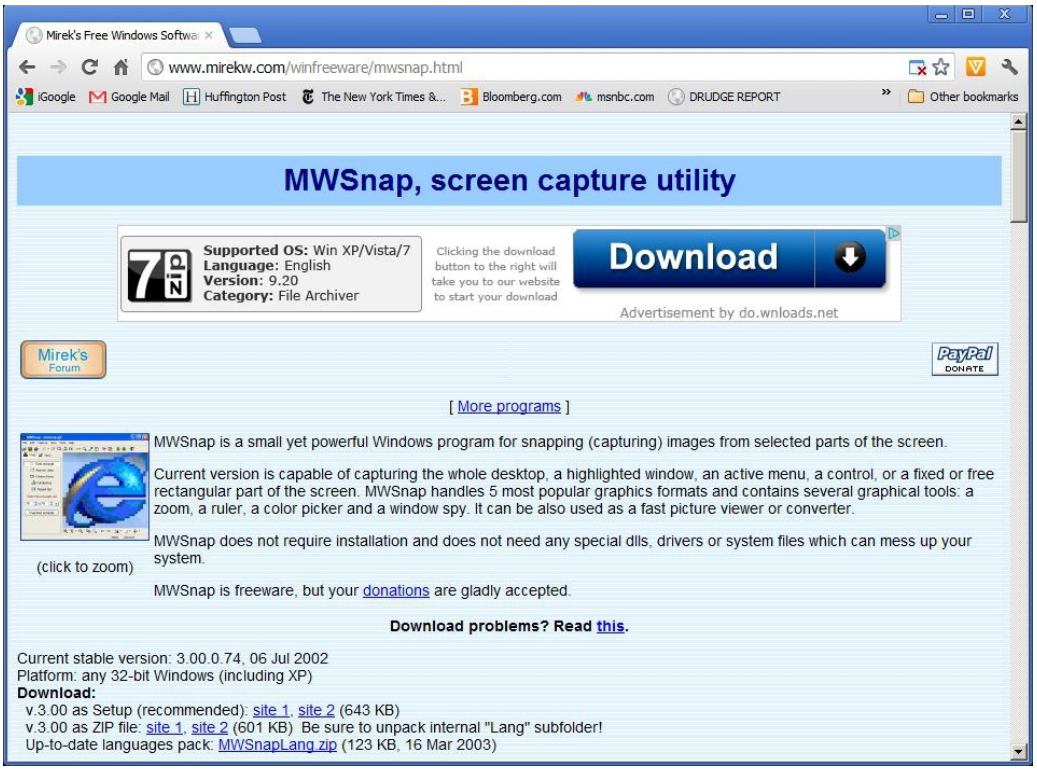


Figure 77: MWSnap Web Site Image

Figure 77 is a screen image of the **MWSnap** web site. The recommended method is to download the installer by clicking on "**v.3.00 as Setup (recommended): site1, site 2 (643 KB)**". Click on one of the two sites to start the download.
When your browser has finished with the download, simply "open" it to start the installer. **MWSnap**

works for all versions of Windows, including the new Windows 8. Just take the default selections on each installation screen.

It is a good idea to create a desktop icon for **MWSnap**, which will allow quick startup of the program. If you select the **MWSnap.exe** file in the Windows Explorer and then right-click, you can select the option "**Send to … Desktop**" as shown in Figure 78 below.

Of course, you can drag-and-drop the **MWSnap** icon to the start menu and be able to start it via that method also.

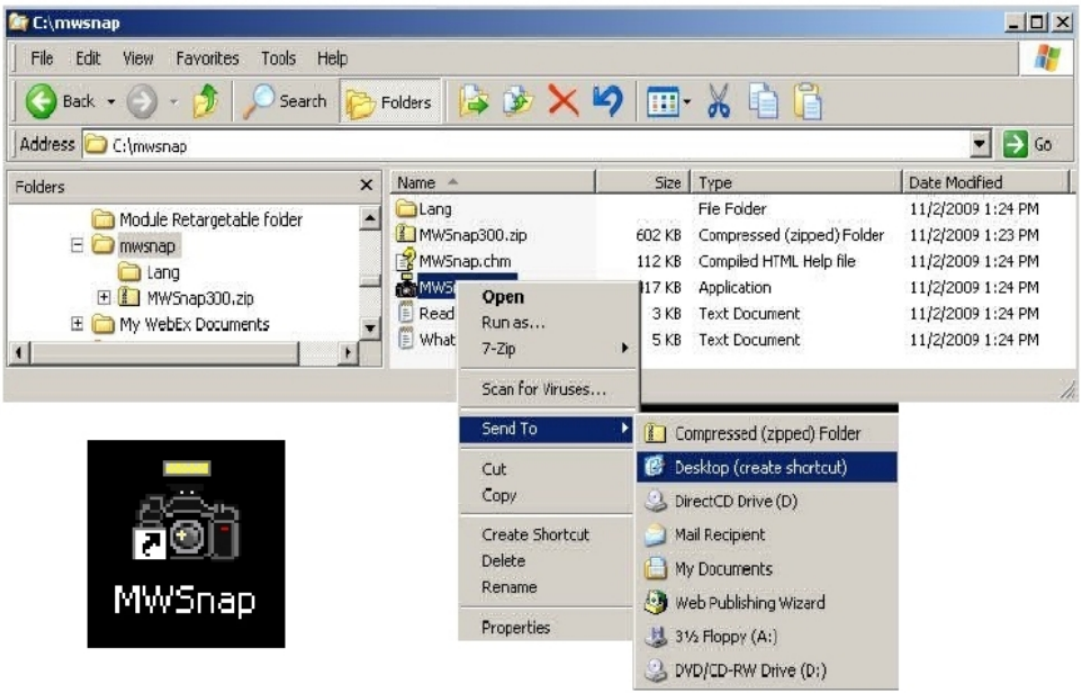


Figure 78: Creating a Desktop Icon

Setting Up MWSnap to Run

Taking a couple of minutes to set up **MWSnap** will yield a screen capture program that is very convenient to use. Setting up the function keys as "hot keys" allows one button activation of area capture, sub-window capture, and full screen capture. Activating automatic file save will record the captured graphic as a jpeg file and make up a simple file name for it.

Hot Keys

Click on the **MWSnap** icon in the task bar or desktop to bring up the default **MWSnap** window, as shown in Figure 79 below. Click on "**Tools - Hotkeys**" to bring up the hot key setup

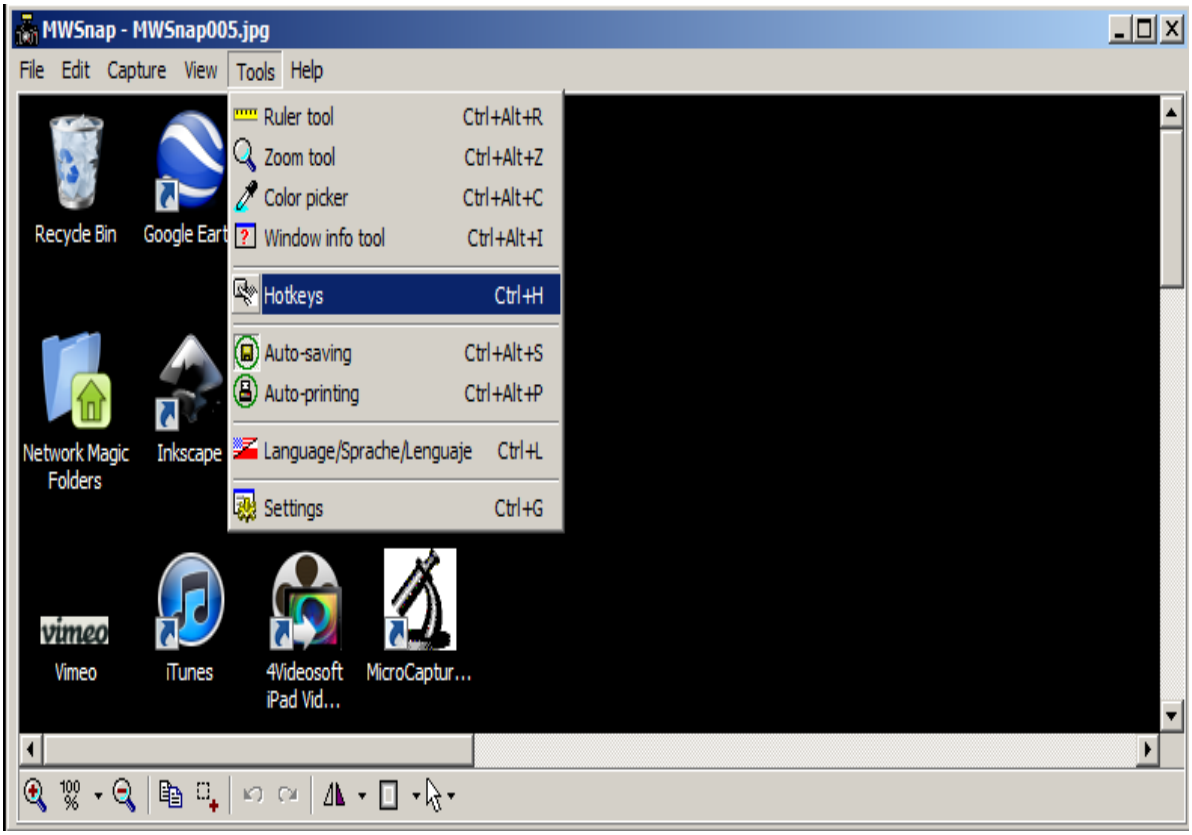


Figure 79: Default MWSnap Window

Note how in Figure 80 below, we can redefine the hot key sequence for "any area", "Window.menu", and “Full Desktop". These will be redefined as follows:

F4	Any Area
F5	Any Window or Menu
F6	Full Screen

Use the pull-down menus in the "Keys" column to select the **F4, F5**, and **F6** keys to act as your "hot keys" for those operations.

Be sure to un-check the **Ctrl**, **Alt**, and **Shift** key assignments presented as the default hot keys for area capture, window capture , and full screen capture.

Also insure that the "**Active**" check boxes are checked. Click "**Apply**" followed by "**OK**" to register your modifications and exit.

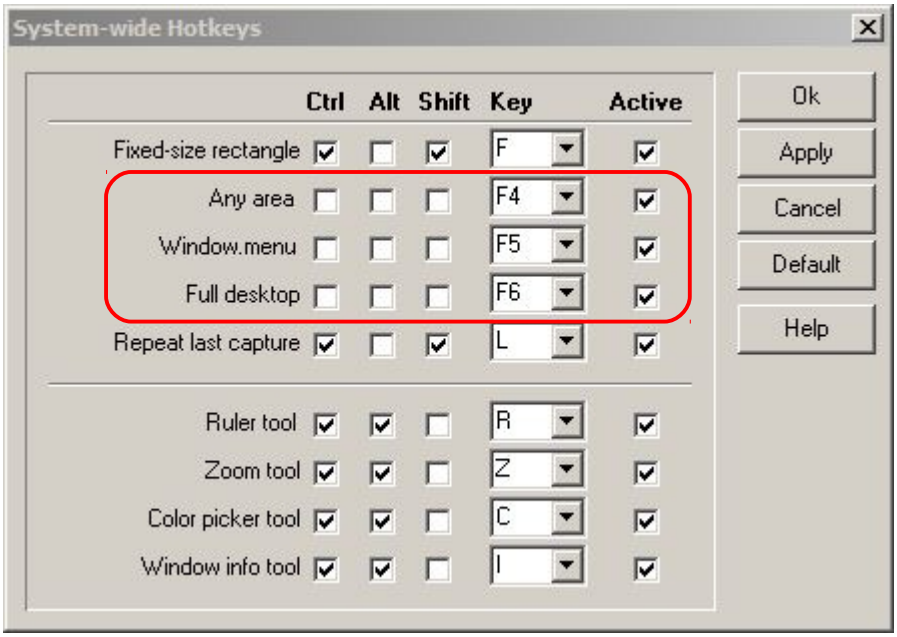


Figure 80: Redefining Some Hotkeys

Automatic Saving

When we "capture" a portion or all of a screen, we'd like this captured data to be saved as a jpeg file automatically with a generated file name. To set this up, enter the Settings menu by clicking "**Tools - Settings**" as shown in Figure 81 below.

Figure 81: Entering the "Settings" Menu

When the Settings window is presented, select the "**Auto-saving**" tab as shown in Figure 82 below.

First, select the Format to be "**jpeg**" (this will save disk space).

Second, check the "**Auto-saving is active**" box to turn on this feature.

Third, enter a folder name and path in the "Save to folder:" text box. In this example, I entered the directory "**c:\capture**" as the destination folder.

Note that the generated file name will be "**MWSnap001.jpg**" incrementing eventually to "**MWSnap999.jpg**".

You can, of course, change that or elect to have the program prompt you for a file name every time you capture something.

Click on "**Apply**" followed by "**OK**" to exit.

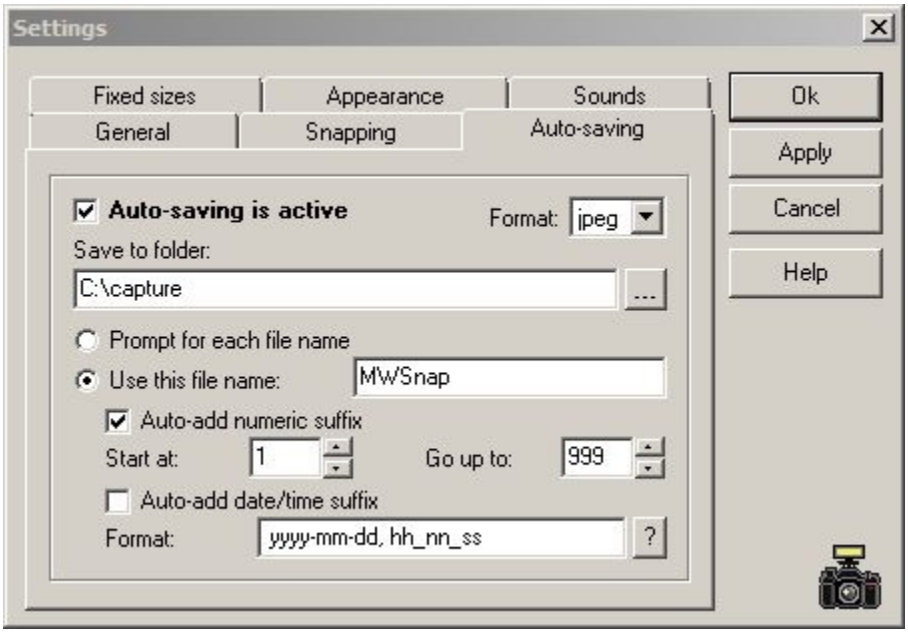


Figure 82: Setting up Automatic File Save on Capture

These setups make **MWSnap** very easy to use. Just run **MWSnap** and minimize it to the bottom task bar. Then hit either F4, F5, or F6 to start up area capture, window/menu capture, or full screen capture respectively. After doing this, use the mouse to select the area or window to be used and click the left mouse button to capture the graphic. The captured graphic will be automatically converted into a jpeg file and placed into your capture storage folder.

By spending just a couple of minutes setting up **MWSnap**, we have reduced screen capture to just a couple of keystrokes (F5 function key ----> move the mouse ----> click the mouse).

Capturing an Area

Assume, as a whimsical example, that we wish to capture the lovely Jennifer Garner and also cut out the somewhat puzzled girl on the right, as shown in Figure 83 below.

Figure 83: Jennifer Garner at the Golden Globes

To do this rather Stalinist bit of editing, first make sure **MWSnap** is running and minimized. Usually you can see the **MWSnap** icon at the lower right of the screen (looks like a camera).

Hit the **F4** function key. The mouse is now used to create a box around the desired capture area, as shown in Figure 84 below. When satisfied with the positioning of the box, **click the left mouse button** to perform the capture. The resulting “cropped” image is in Figure 85.

Sometimes the **MWSnap** default window pops up after a capture, just minimize it if that occurs. If this really irritates you, select "**Tools - Settings**" and under the "**Snapping**" tab, remove the check mark from the option "**Restore MWSnap after Snapping**" to prevent this pop-up from appearing.

Figure 84: Capturing an Area with MWSnap

Figure 85: Cropped Image

As can be seen in Figure 85 above, the jpeg captured graphic has the cropping we selected with the mouse. It's a file with a name like ***MWSnap001.jpg*** which you can insert into documents and emails.

Capturing a Specific Window

To capture a specific window on a display with many windows, use the **F5** function key. As you move the mouse around the screen, **MWSnap** will show you which window it will capture by drawing a perimeter or border around the window. When you have the desired window outlined, click the mouse button to capture.

As a practical example, let's capture the COM6 Serial Print window running with the Arduino IDE as shown in Figure 86 below.

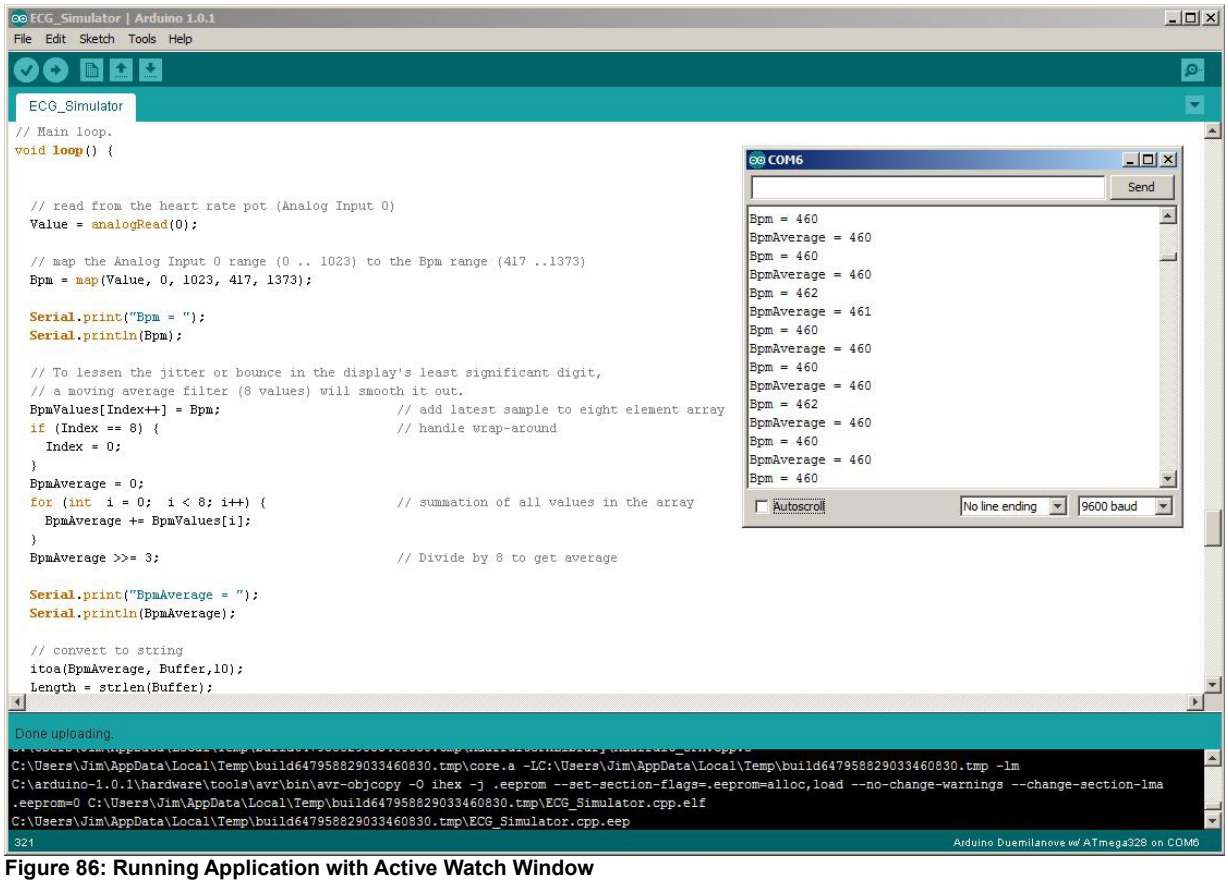


Figure 86: Running Application with Active Watch Window

To capture only the Serial Print window, hit the **F5** function key to start window capture. As you move the mouse around the screen, **MWSnap** will indicate which window to be captured by drawing a border or perimeter around the window selected. Once satisfied with the window selected, **click the left mouse button to capture**.

Everything within the perimeter will be captured, which means that sub-windows within the captured window will be part of the image. Figure 87 below shows the perimeter that **MWSnap** draws around the Watch window.

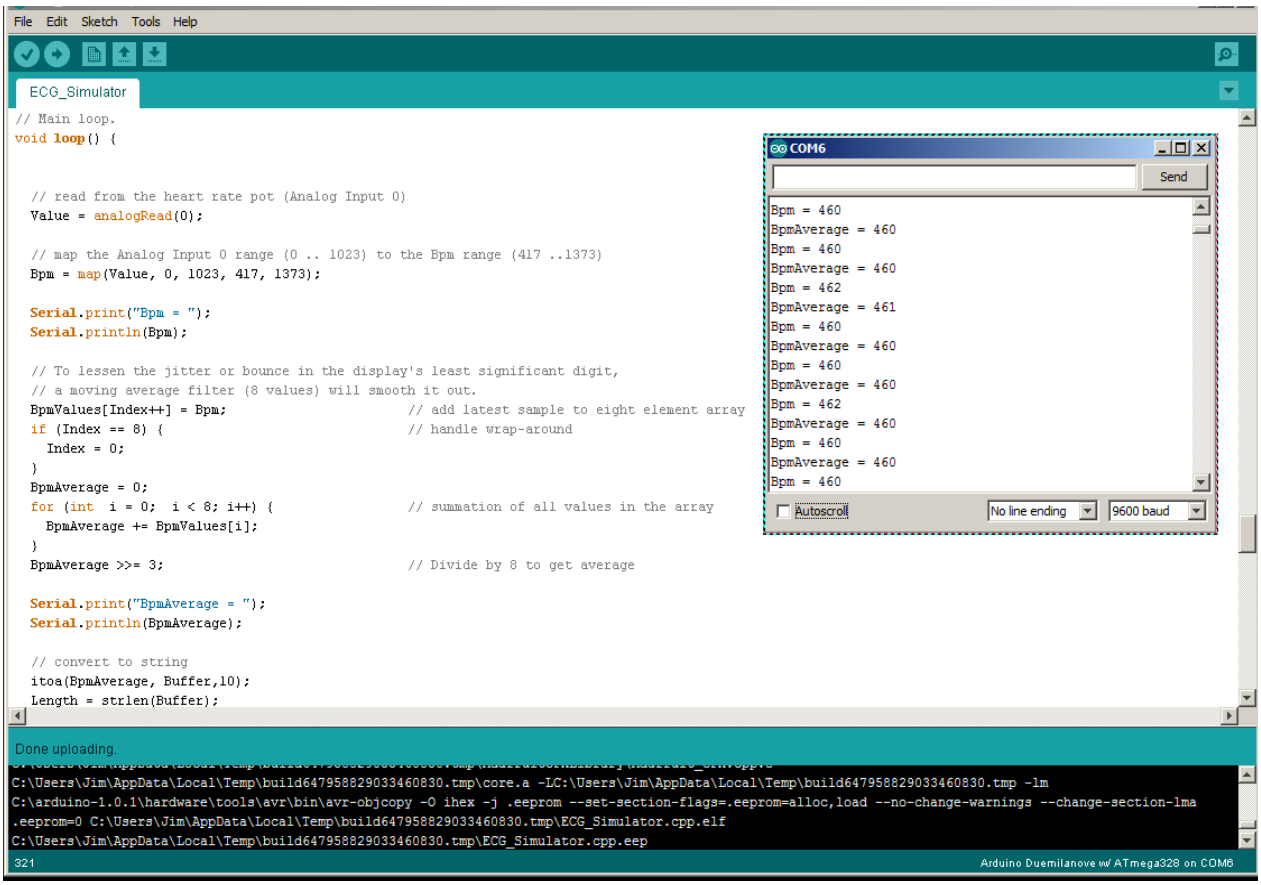


Figure 87: Hit F5 function key, Move mouse until Watch window is Selected, Click to Capture

Figure 88 shows the captured window image. It will be a jpeg file with a filename such as **MWSnap002.jpg** and will reside in the **c:\capture** folder.

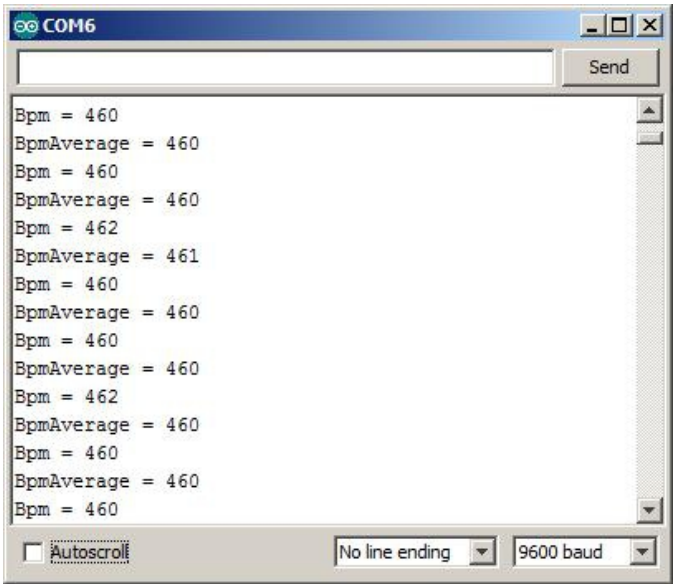


Figure 88: Captured Watch Window

Capturing a Pull-Down Menu

Capturing a pull-down menu is a challenge for these kinds of programs. I've seen others do it by a delayed capture; some programs can't do pull-down menu capture at all.

MWSnap makes this operation very easy. Basically you use the mouse to activate the pull-down menu and then hit the **F4** (area) or **F5** (window) function key to start the operation. The mouse is now under the control of **MWSnap** and you can set a perimeter around the pull-down menu and then **click to capture**.

As a simple example, suppose I wish to capture the Arduino “Tools – Board” sub-menu as shown below in Figure 89.

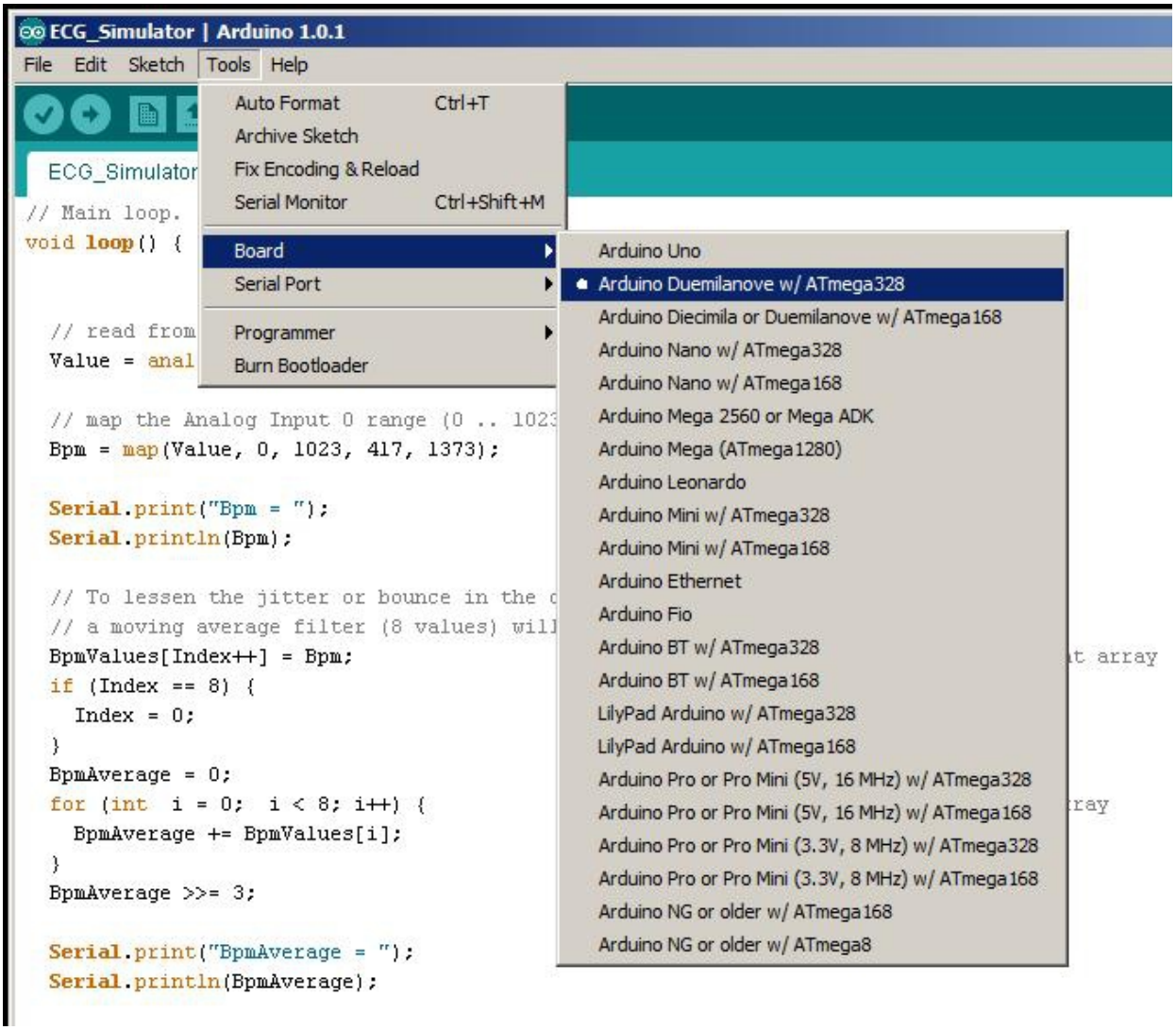


Figure 89: Capturing a Pull-down Menu

In this example, use the mouse to bring up the Board pull-down menu as shown in Figure 89 above. Then hit the **F4** (area) function button and use the mouse to draw a perimeter around the Project pull-down menu as shown in Figure 90 below. When satisfied with the capture area, **click the mouse to capture** the image.

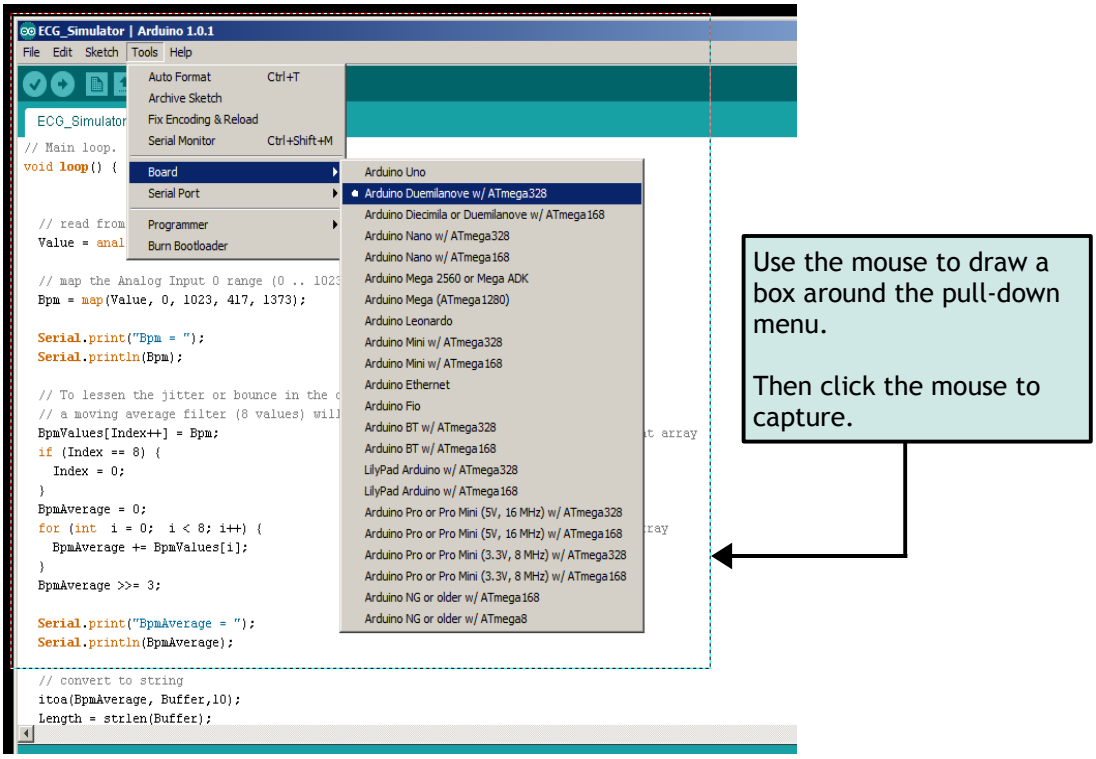


Figure 90: F4 (area) Used to Outline the Pull-down Menu

The captured image is shown in Figure 91 below.

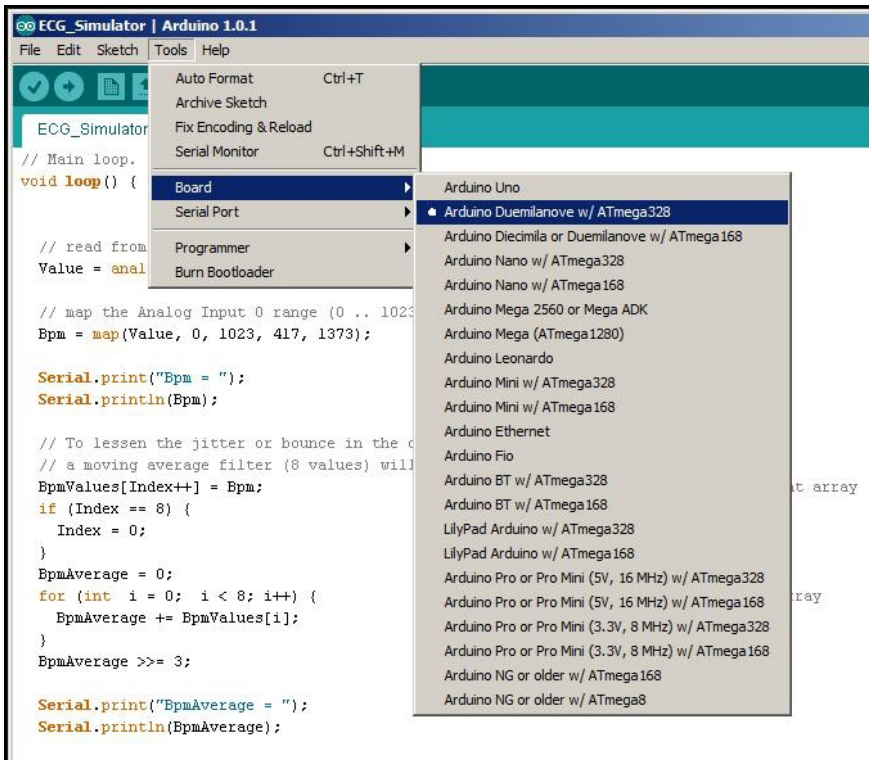


Figure 91: Captured Pull-down Menu Image

Full Screen Capture

To capture a full screen, just hit the **F6** button. The entire screen will be captured, converted into a jpeg file, and written to the capture folder with a filename like "MWSnap004.jpg". Figure 92 shows a full screen capture.

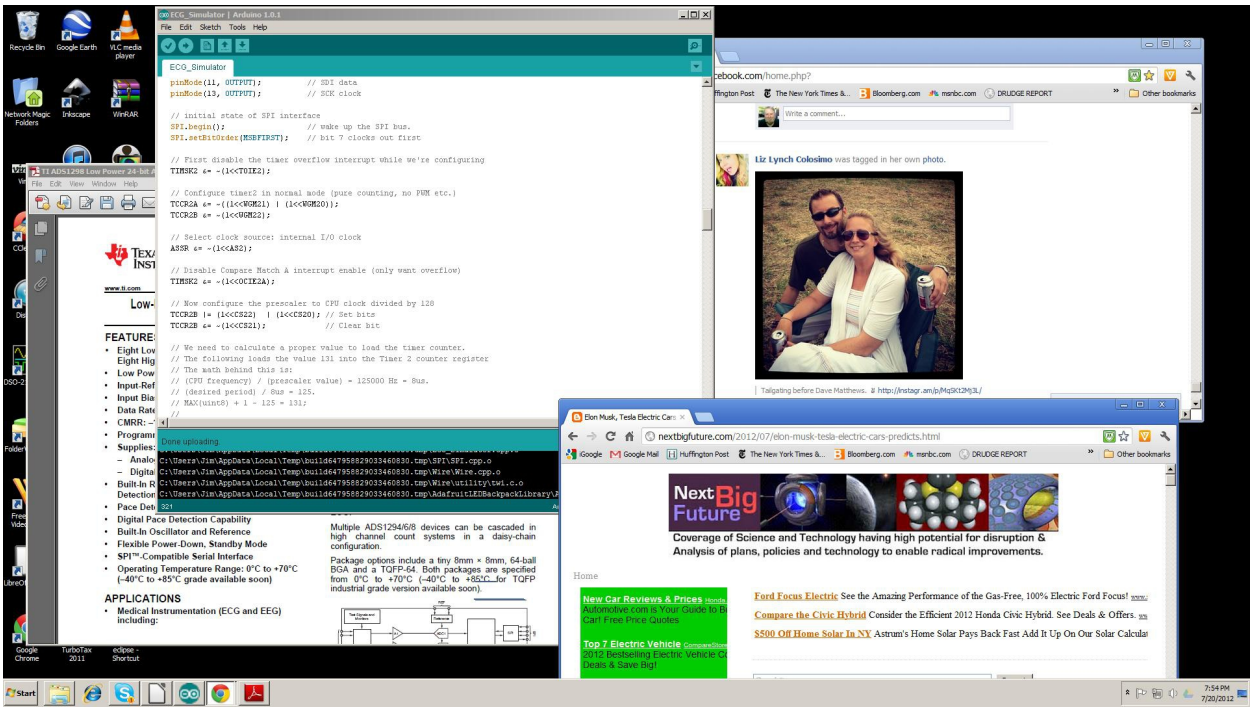


Figure 92: Full Screen Capture (F6)

Other MWSnap Features

This report only covers the bits of **MWSnap** that are essential for creating documents with screen images; area, window and full screen capture.

There are many other features to explore. You can peruse the rather extensive help file to do this.

For example, there's a fixed rectangle snap (several sizes are available) that you can easily maneuver around the screen and capture whatever is currently within the rectangle (Figure 93).

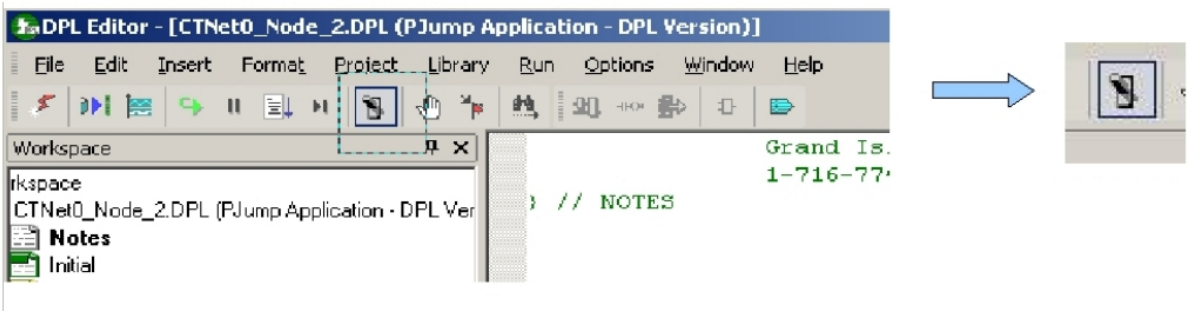


Figure 93: Fixed Rectangle Capture

MWSnap also has a unique pixel ruler which you can position around the screen as shown in Figure 94. The ruler can be switched to vertical orientation or have it's marking direction reversed.

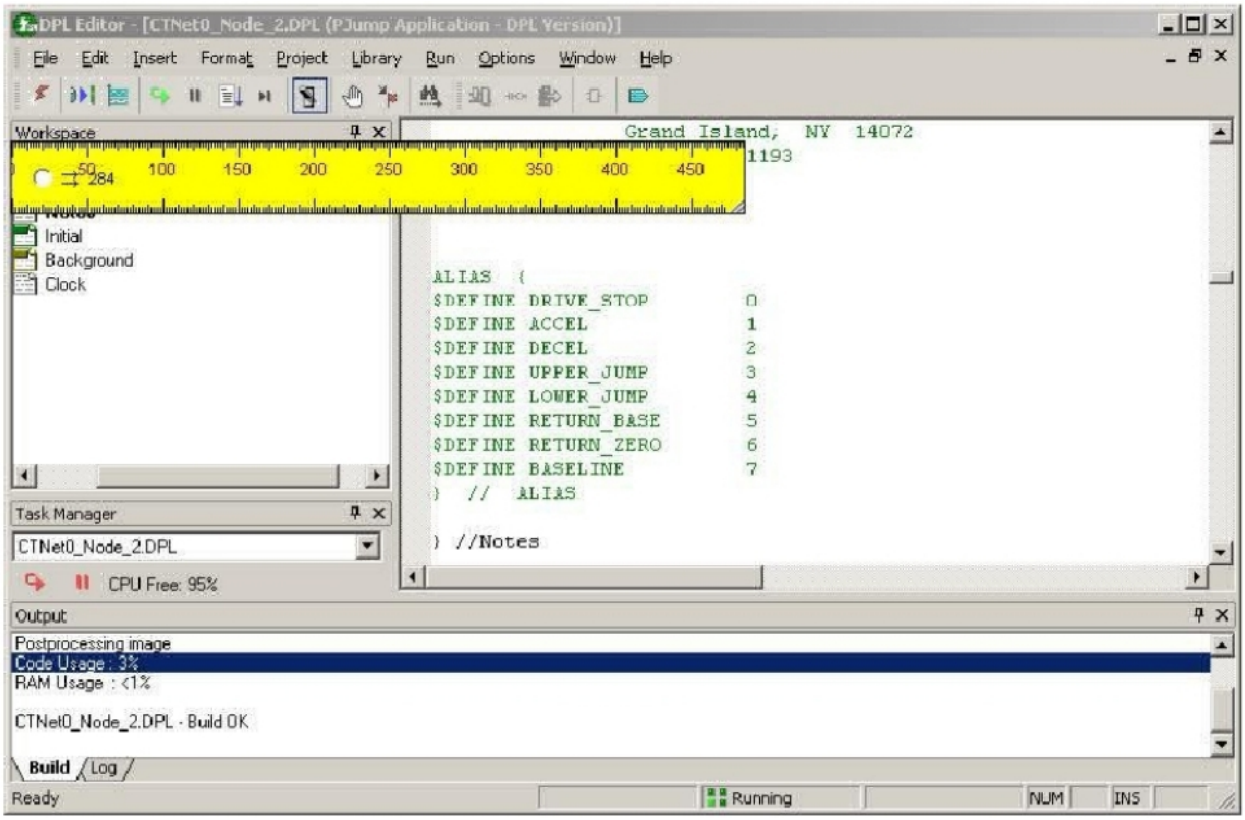


Figure 94: MWSnap's Pixel Ruler

MWSnap has a magnifying glass. Just click on the magnifier and move the cursor around the screen. A zoom window will appear and show a magnification of the spot you are presently at. You can change the zoom with the **+** and **-** keys (click within the zoom window to do this); you can make a bigger zoom window with the **ctrl +** and **ctrl -** keys. Figure 95 shows this feature in operation.

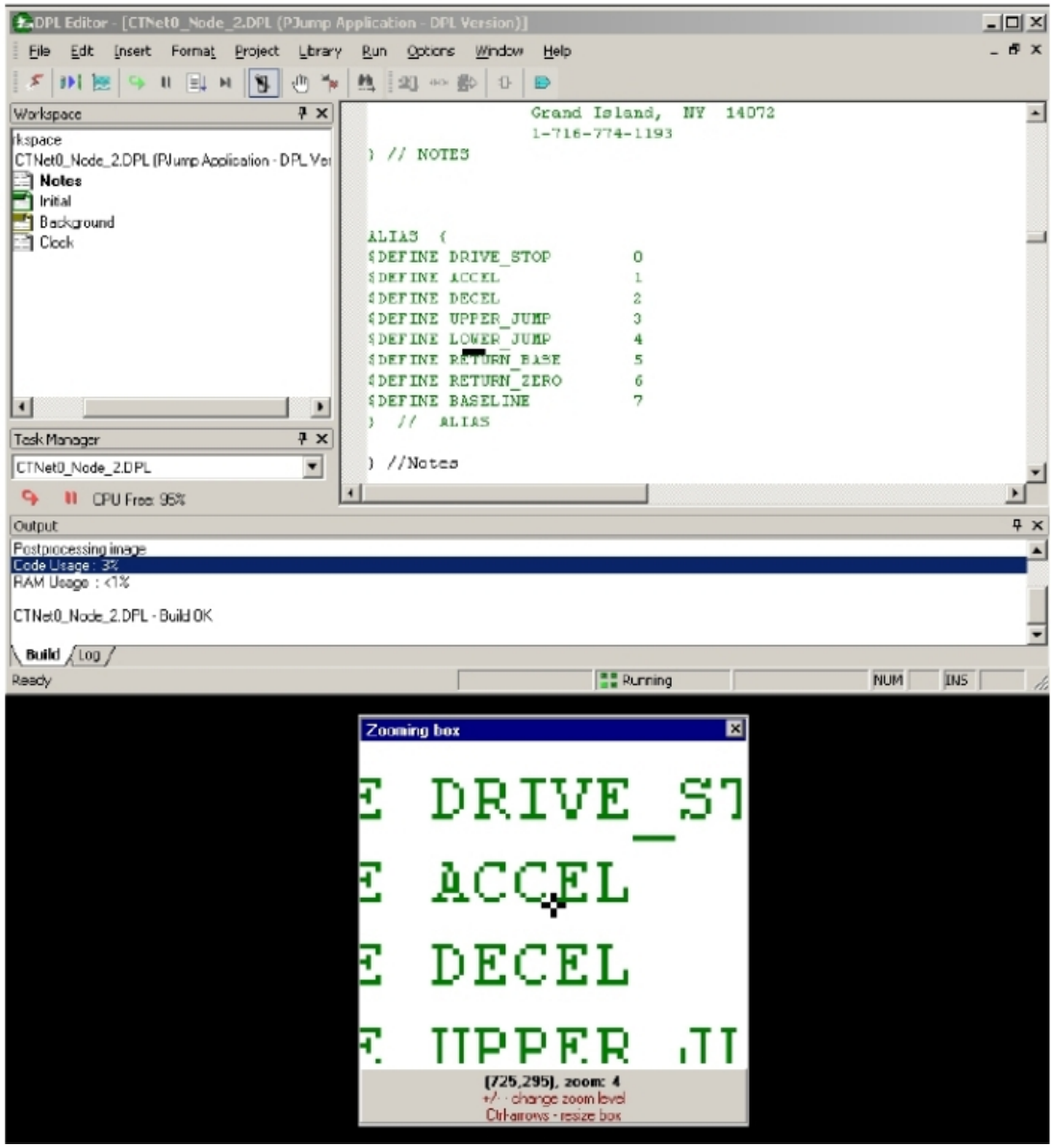


Figure 95: Using the Zoom Magnifier Tool

Conclusions

This is one of the best freeware programs I have ever used. Set up to work with function keys, screen capture is quick and efficient with a minimum of key strokes. **MWSnap** works with just about all Windows versions (it works with my copy of Windows 8). Feel free to give it a try (an obscure pun).