A Magnetic Loop Antenna and Controller for 40 Meters
John Price – WA2FZW

License Information

Introduction

After a long hiatus from ham radio, at the urging of some old friends, I decided to get back into it. My previous activities in ham radio were mostly on VHF and I was seriously into building my own equipment.

This time, I decided to venture into the world of HF and QRP. So I purchased a kit for a transceiver known as the BitX-40. This little radio has a nominal output power of 5 to 7 watts, and a pretty decent receiver. It is controlled by an Arduino Nano micro-controller.

After building the kit and making a number of modifications to it to add capabilities, the next project was an antenna. I'm getting to old to be climbing on the roof, so I looked for something that could be installed in the attic. I decided to try a pair of "Hamstick" antennas in a dipole configuration. Well, it worked, sort of; the fact that it is a pretty inefficient antenna to begin with and the fact that it was hanging horizontally four feet above a sea of foil backed insulation made it even worse.

Although, I managed to make a few contacts when band conditions were really good, it quickly became clear that I was going to need a better antenna system. A friend suggested that I look into magnetic loops, which I did. The first criterion was, would it fit in the attic. The answer to that was yes, so I decided to give it a try.

The one problem with this type of antenna is that the bandwidth is extremely narrow. It is basically a high-Q L-C circuit, and thus needs to be dynamically tunable as one changes frequencies by more than a few KHz.

More research on the web turned up any number of Arduino based tuning arrangements for these antennas One of the best articles I found is one by Loftur Jónasson - TF3LJ / VE2LJX. His controller is pretty slick, but has a lot of bells and whistles that I didn't think were necessary for my application. I did purchase one of his circuit boards as a "Plan B" if my design didn't work out. I also tried to use parts of his design, but unsuccessfully; more on that later.

So, I decided to attempt to build my own antenna and controller from scratch. I will describe the three main components of the system:

- The antenna itself
- The controller hardware
- The controller software

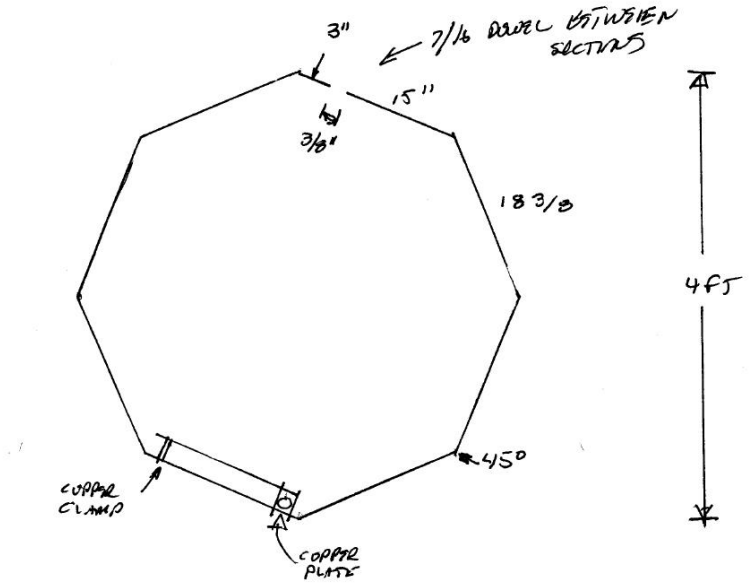
I will not only describe the finished project, but will describe things I did wrong and where I see room for improvement.

The Antenna

Initial Design

The design of the antenna itself went through a number of significant changes after I started this project. The basic design criteria was that it had to fit in the attic, thus the overall diameter had to be about four feet, which, by the way is less than the optimum size for a 40 meter loop.

This is the original sketch of the basic design, which doesn't show the tuning arrangement, as I hadn't figured that out yet:



The loop is constructed of ½" copper pipe with 45° elbows at the corners. The octagon shape was selected basically for ease of construction. Also, as shown, my intent was to use a gamma match arrangement as opposed to the more conventional coupling loop for these antennas. I found that design on the web in an article by AA1IK.

The Tuning Mechanism

As I stated in the introduction, one of the characteristics of loop antennas is that they have a very high "Q", and therefore must be retuned whenever the operating frequency is changed by more than a few KHz (the actual bandwidth varies with the design). This requires a tuning capacitor across the gap shown at the top of the initial design drawing shown above.

That capacitor has to have significant voltage handling capability if the antenna is to be used with any kind of high power transmitter. As I'm only dealing with a few watts, it's not a large consideration, but I figured it would be better to design and build for the eventuality that I might someday want to use it for more than the QRP radio.

The most common capacitor that people use are vacuum (really inert gas filled) variables. I managed to find a couple of old Jennings vacuum capacitors on eBay for a rock bottom price; they generally run around $150 each; I got two for $40.

The only problem was that one of them was missing the adjustment shaft, and the shaft on the other had a gear attached to it that I didn't need or want, and I could not figure out how to remove (neatly). Solution: find some threaded rod of the right size/pitch to replace the stock shafts.

Well, that didn't happen! The shafts are ¼" diameter, but they do not have any standard pitch that I could find. They weren't 24 or 28 threads per inch, which are American standards. Someone from England on one of the internet groups suggested that they might be an old British standard known as "British Standard Fine", which has a 26 thread per inch pitch. Nope! After actually finding some BSF threaded rod and waiting a few weeks for it to arrive from Brittan, that wasn't the right pitch either.

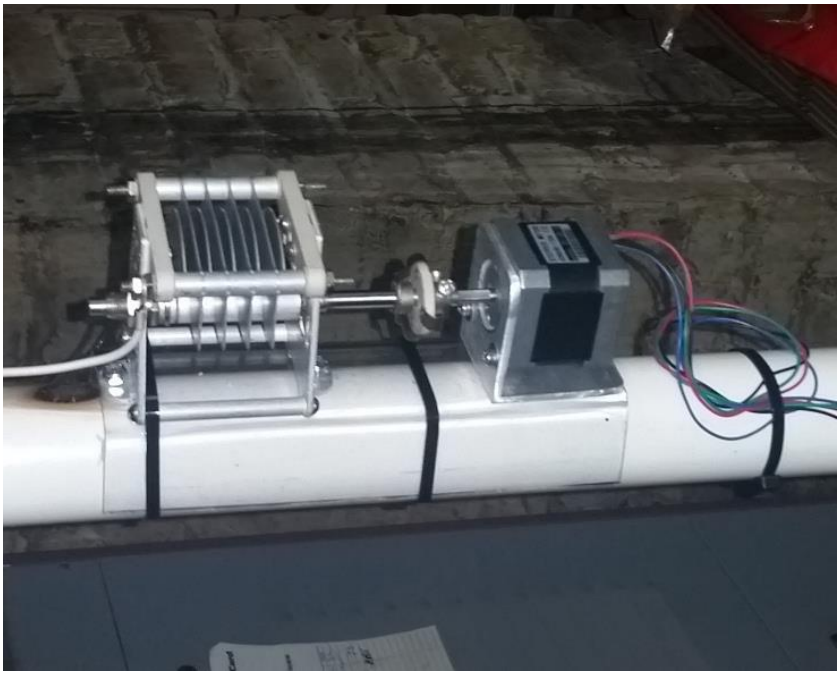
As luck would have it, I found a 4.5KV air variable capacitor at a local hamfest for a reasonable price, so I decided that that would suit my needs just fine.

Next issue is how to adjust the tuning. Going up to the attic to move the capacitor every time I wanted to move to a new frequency obviously isn't a good solution, so I was going to need some kind of motor that could be operated remotely to handle the task.

Enter the stepper motor. If you're not familiar with these, they can rotate in very small, precise increments. They are often used in such everyday devices as printers, DVD players and hard disk drives where precise positioning of a component (print head or disk head) is required.

Great! That problem is solved, but now we need to figure out how to actually control the motor. That will be discussed further detail down the page.

Here's what the motor/capacitor combination looks like mounted on the antenna support mast:



Two things to note; I mounted the capacitor and motor on a piece of Lexan so they are insulated from each other, and the shaft coupler has a ceramic insulator built in. Again, the voltage on the capacitor can be quite high, so this is an important design feature.

The motor, by the way, is an Oukeda 42HS40-1704 (NEMA Size 17, 1.8 degree, 2 phase) motor. They can be found on eBay from several Chinese sources rather cheaply.

Design Evolution

Once I got to the point where I could tune the antenna remotely (using the Arduino Uno and some very basic software), I started playing with the actual antenna design.

More reading on the internet suggested that adding loops to the antenna would increase the efficiency by a factor equal to the number of loops, up to 3 or 4 loops. Since I already knew that the 4' diameter was below the recommended minimum size for a 40 meter loop, I decided to add a 2nd loop.

I added the 2nd loop. The spacing between the 2 loops was fixed by the diameter of the 1 ½" PVC pipe used as a support mast which is pretty close to what is considered to be the optimum spacing between the loops. Short pieces of ½" PVC pipe with the ends rounded out to match the radius of the ½" copper pipe are spaced around the loops to maintain the spacing. Rather than using screws to affix the spacers to the loop itself, I simply put nylon wire ties around both loops at each spacer, as shown here:

So far, so good, except that after adding the 2nd loop, the antenna would no longer tune. Adjusting the length of the gamma match, had little or no effect. After a lot of head scratching, I realized that after I added the 2nd loop, the gamma match was no longer located at the center point of the antenna, but rather at the ¼ or ¾ point, depending on where you started measuring from.

I didn't want to move it to the top of the antenna, as I felt that it would be too close to the tuning capacitor, and thus maybe it would adversely affect that part of the tuning system. Instead, I scrapped the gamma match and went with the more conventional coupling loop, as shown here:

The coupling loop is approximately 11” in diameter, or about 22% the diameter of the main loop. Articles on the internet suggest that the coupling loop size should be about 20% of the diameter of the main loop, so I’m pretty close.

I used an MFJ antenna analyzer to tune the coupling loop. I set it for a 1.0:1 SWR at 7.150 MHz. It’s a tad higher at each end of the band, but not enough to matter.

The distance from the bottom of the loop to the top of the main loop tubing is about 2.5”. The distance makes a difference in tuning the coupling loop (notice all the holes drilled in the mast to adjust the position).

One change I want to make is to flip the coupling loop over so that the coax connector is at the bottom.

Here’s a close-up of the tuning arrangement for the coupling loop:

The standoff is the type with a built in stud on one end and the other end tapped for a screw. I drilled a 1/8” hole about halfway along the length to accommodate the 1/8” copper tubing used for the coupling loop. Loosening the screw allows the circumference of the coupling loop to be easily changed. This would have been easier if I had used the drill press at my shop, but I did it using a hand drill, so it can be done that way.

Here’s the finished antenna in the attic:

The Controller

Design Objectives

What I considered as required design objectives for the Arduino controller were:

- Must be able to tune the antenna automatically on command (as opposed to continuously tuning based on frequency).
- Must be able to manually fine tune the antenna via some sort of rotating control.
- Must display the SWR somehow.
- Must display any actions being taken by the controller as it performs its duties.

The Main Components

I selected the Arduino Uno processor as the heart of the controller (mainly because I already had one lying around).

To operate the motor, I'm using a DFRobot DRI0023 (V2) shield. These are again, available from any number of Chinese sources on eBay. Note that the Version 2 shield has the ability to rotate the motor in more partial steps that the Version 1 shield, and more importantly, it provides the capability to disable the motor when not actually being used. I found this to be a critical feature, because for whatever reason, simply having the motor enabled (not actually turning) was creating a lot of noise in the receiver. Even though the controller is running off a separate power supply from the radio, having the motor enabled still makes noise in the receiver.

The manual tuning control is a Keyes KY-040 Rotary Encoder. As is the case with almost everything else used in this project, these are available on eBay from guess who for about a buck. Make sure you get one with the built in push-button switch. You'll learn why later.

The next piece of the controller is a FICBOX IIC/I2C 1602 2 line by 16 character LCD display. One thing to note here is that the particular display that I got has a non-standard address. The normal default address for an I2C display is 0x27; this one has an address of 0x3F. It can be changed by wiring jumpers on the circuit board.

Should you have trouble with the Arduino not seeing the I2C device, there are a number of sketches on GitHub that will scan your system and report any I2C devices they find. I would provide a specific link to one, but I can't remember which one I used!

The final piece needed is a device to tell the controller whether or not the antenna is actually tuned.

My first idea was to build a field strength meter into the antenna itself, figuring that radiated power would be a better indication of proper tuning than SWR, as that depends not only on the antenna, but possibly the length of coax between the transmitter and antenna as well. That didn't work because of the amount of cross-induction between the wires coming from the measuring coil and the noise created by the motors on the wires for that, which are in the same cable. I might have been able to solve that by running a separate shielded cable for the measuring coil, but didn't particularly want to snake another cable between the basement and the attic.

Thus, I decided to use some sort of SWR bridge to indicate proper tuning.

In one of Loftur's presentations about his controller, he talked about using a SWR bridge kit from an outfit in Florida, KitsAndParts.com. The SWR bridge is item #19 on the list of stuff available as of the time I write this. And, by the way, this place is also a great place to buy toroid cores; as a matter of fact, I'm told, the toroids in the BitX-40 came from this place.

I didn't have much luck with the SWR bridge kit. It seemed to work fine with a voltmeter watching the forward and reflected voltage, but when I connected it to the Arduino, the readings I got on the analog input pins were all over the place; seemingly random numbers. Loftur used an AD8307 logarithmic amplifier to boost the voltage levels coming from the bridge. I didn't have much luck with that either (but I'm still working on it).

What I did make work was tapping into a plain old cheap Astatic PDC1 SWR meter to get the forward and reverse voltages. You can find these on eBay, Amazon and a number of other places for less than $20. Although the voltages from this unit were less than those from the little kit bridge, when connected to the Arduino, the readings on the analog inputs were stable and reliable. I was concerned by the fact that generally the reading I get on the Arduino reverse voltage input pin is less than 50, which I thought would be too small to accurately tune the antenna. The maximum reading possible is 1023. But, again, it works, so no need to fix it, although if I get the AD8307 amplifier working, it probably can be improved.

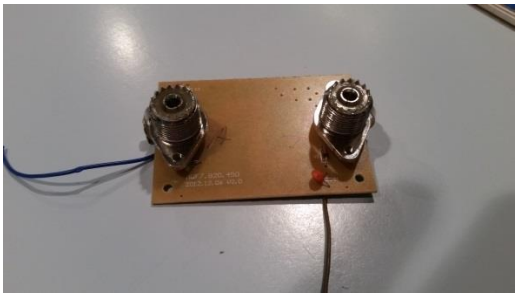
Controller Construction

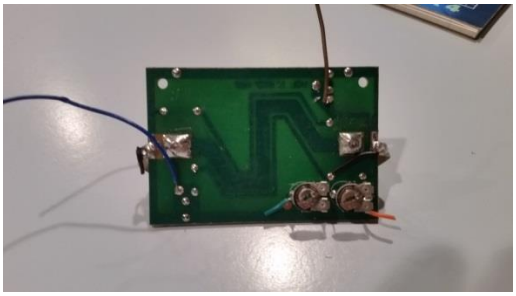
The five main components of the controller are the SWR bridge, the Arduino Uno processor, the DFRobot DRI0023 (V2) shield, the LCD display, the rotary encoder and a homemade board to interconnect all the other parts.

The SWR Bridge

After playing around unsuccessfully with the SWR bridge kit from KitsAndParts.com, I decided (at least for the time being) to use the SWR bridge circuit board from the Astatic PDC1 SWR meter to get the forward and reverse voltages. In testing, it worked, so, why not?

Initially, I just tapped into the PDC1, leaving it otherwise intact. That gave me a convenient analog meter to allow comparisons to what I was seeing on the Arduino input. In the final construction, I removed the SWR bridge circuit board from the unit and built it into my enclosure. Here's what the board looks like:

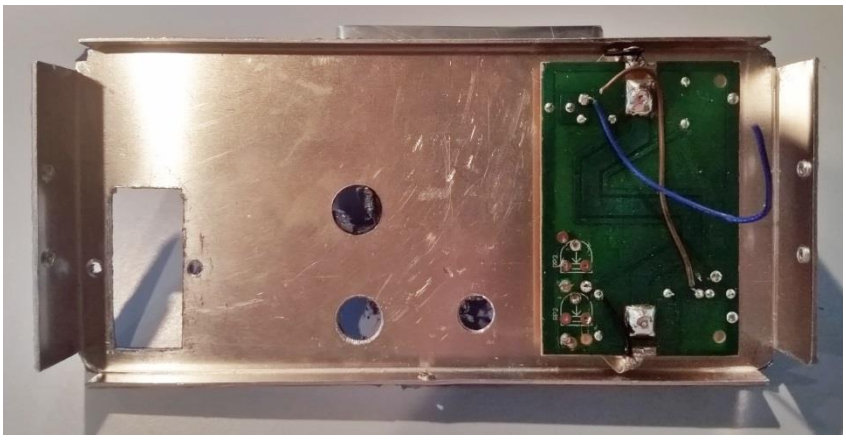




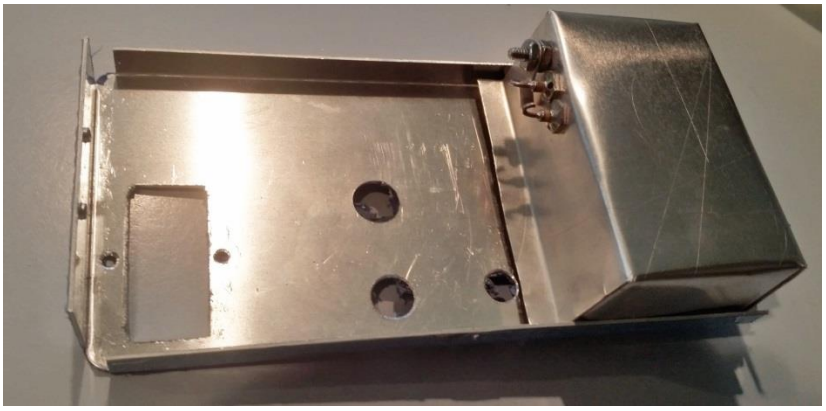
It’s a pretty basic circuit. On the right hand picture, you can see that the transformers are etched into the PCB, and at the ungrounded end of each, there is a resistor, diode and capacitor. The arrangement is symmetrical; in other words, either SO-239 could be used for either the transmitter connection or the antenna connection. If connected as originally marked on the rear of the PDC1, the blue wire is the reverse voltage, and the brown wire is the forward voltage.

The trimpots shown in the right hand picture are unused here, and were removed.

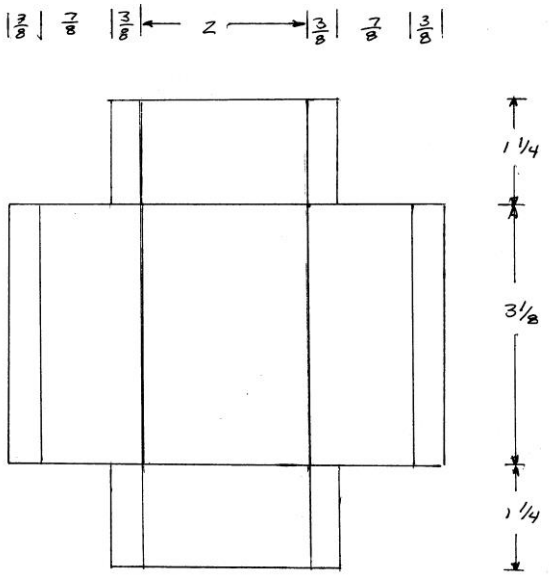
Here’s the SWR board mounted on the back panel of my enclosure:



During breadboarding and testing, I had some issues with RF noise getting into the receiver, particularly from the motor driver. To alleviate that, I put a shield around the SWR board, as shown here (just fitted in place for now):



Note the use of feedthrough capacitors between the RF section and the rest of the controller enclosure. Here's the template I used to make the shield enclosure:



The layout shown just fits snugly around the SWR board, and in my case just snugly inside the enclosure. In the enclosure that I used, there is only about 3 1/8" of vertical space and the SWR board and shield fit very snugly. I thought about mounting the board horizontally, but opted not to; no particular reason.

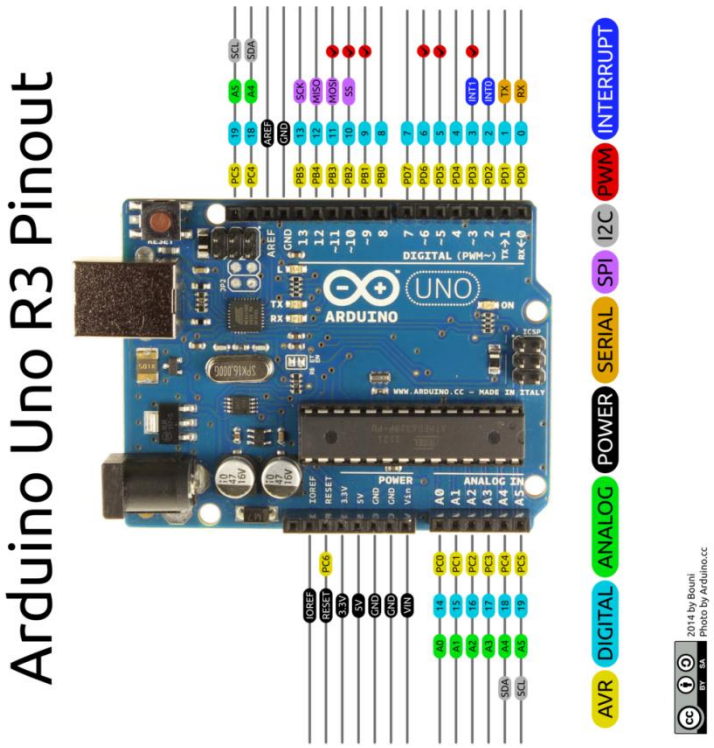
Motor Control Shield

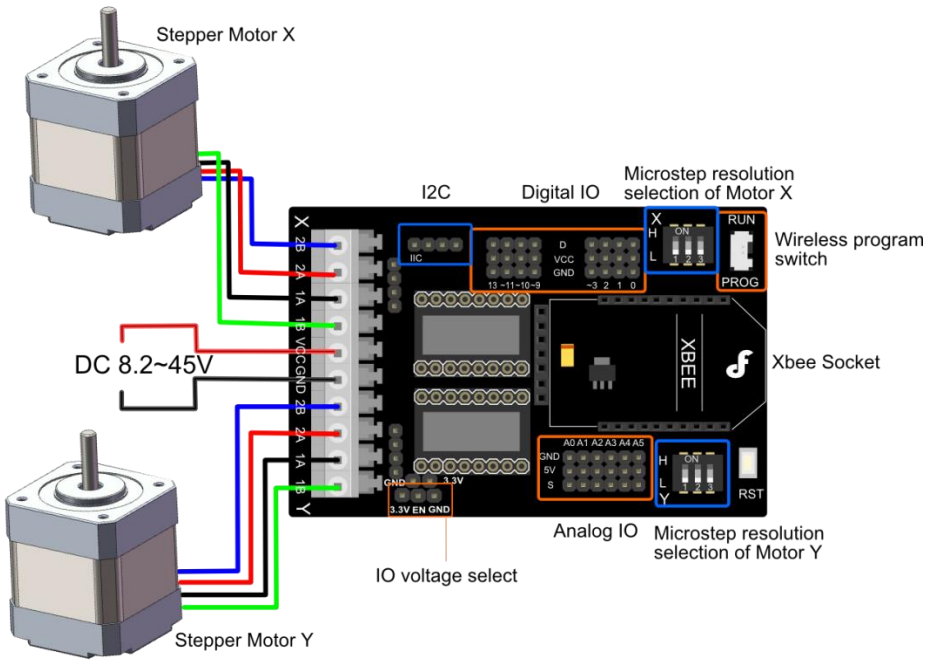
The fact that the motor controller is called a *shield* means that it is designed to plug directly into the headers on the Arduino board without any need for any additional wiring between the two components.

This has advantages and one slight disadvantage. The advantage as stated above is that there is no need to do any additional wiring to make the two devices work together.

The one disadvantage is that the shield determines which digital and analog I/O pins are going to be used for the two units to communicate. In the case of my antenna tuner, only one motor is used, but the Arduino I/O pins associated with the 2nd motor become unavailable for other uses, unless one uses wire cutters to remove the pins from the motor shield.

Here are pictures of the processor and the motor shield showing the pin configurations of both devices. I rotated the picture of the Arduino sideways so you can better see how the pins line up between the two.





The Interconnect Board

The interconnect board sandwiches between the Arduino board and the motor control shield. Its function in life is to provide a neat way of connecting all of the components together. Without it, due to the way the I/O, power and ground pins are scattered about on the motor shield, there would have been a lot of single wires running from the LCD, encoder and front panel switches and jacks to the appropriate pins on the motor shield.

By creating this interconnect board, there will be only one ribbon cable running from each of the other major components (5 wires for the encoder, 4 wires for the LCD, etc.) to the board.

The board also contains two relays. RY-1 is the transmitter keying relay and RY-2 fixes a problem with the direct keying option

The transmitter keying relay can be turned on and off from a toggle switch on the front panel. It is not needed for the BitX, as it can be keyed directly from the *Xmit_Key* pin on the Arduino. Other transceivers however might need the relay, depending on how the CW keying circuits in those rigs actually work. Specifically, it the keying line has more than 5VDC on it with the key open, you need the relay as putting more than 5V on the Arduino is going to make bad things happen.

One note about the relays; the NTE R74-11D1-5 relays that I used have the spike protection diodes built in, thus there are no external diodes associated with them. If you use this type of relay, it is important to pay attention to which side of the coil goes to the more positive voltage. If you use relays without the built-in diodes, external ones must be added.

Although the BitX can use the direct keying option, there was a problem. If the power to the controller was not on and the front panel switch was in the *Direct Keying* mode, when you turned the BitX on, the transmitter would immediately key, as the *Xmit_Key* pin on the controller's Arduino was in a low state. RY-2 was added to disconnect the *Xmit_Key* pin from the transmitter keying line until power is applied to the controller, while leaving the direct connection between the *CW Key* jack on the front panel and the *Transmitter Keying* jack on the rear panel.

If I ever get the AD8307 amplifier working for the SWR inputs, those will be added to this board as well.

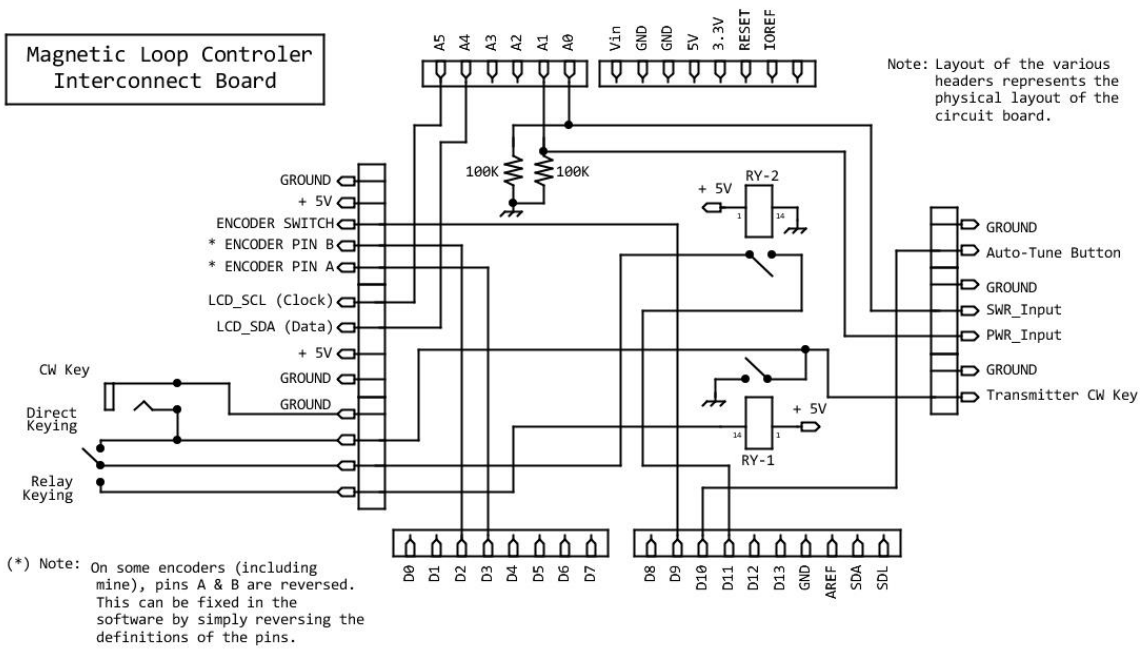
One thing to note, you can't simply use a general purpose prototyping board to construct the interconnect board. It turns out that while the headers for the four groups of Arduino pins have standard 0.1" (2.54mm) spacing, the gap between the analog I/O pins and the power pins is not a standard 0.1" spacing.

Thus, you have to use a prototyping board designed specifically to connect as a *shield* to the Arduino. I found a lot of choices online, and settled on this one as being the cheapest and also void of real estate being taken up by space for things I neither wanted nor needed. Many of the other boards I found had places on the PCB for buttons, LEDs and other things unnecessary stuff. The one I selected is just a plain vanilla field of 0.1" spaced holes.

For now, the connections on the board are simply wired with 30ga wire. Perhaps, at some point in the future I could design a printed circuit board, however, that would be something I've never done before and so that project can wait.

The relays are NTE R74-11D1-5, which is what I managed to find at my local parts supplier. It is, in fact a DPDT relay, but any 5V SPST relay will work.

Here's the schematic for the board. Note, that for clarity, most of the +5V and ground connections from the headers to which the components connect are simply labeled as such, without showing the actual wires.



The two 100K resistors from the SWR and PWR input lines to ground were added late in the construction. They add some stability to the readings on the Arduino inputs.

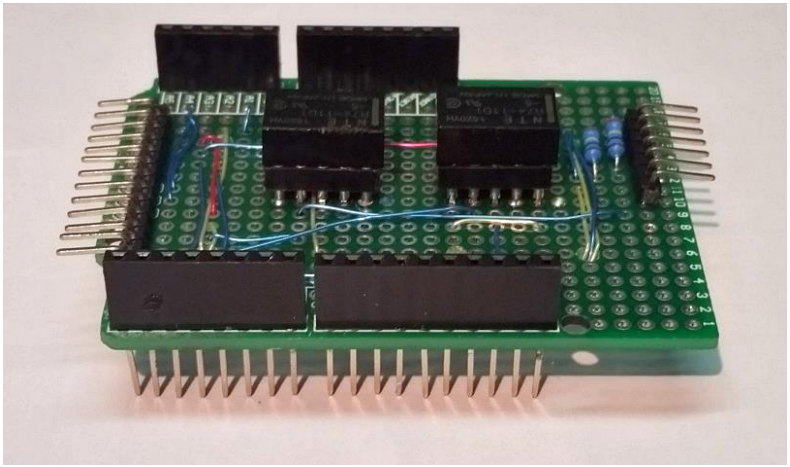
I placed the headers that connect to the LCD, encoder, etc. on the board such that they would be as close as possible to the things they connect to. That might not have been the smartest layout as there are wires running in all kinds of directions on the board. The placement of the headers in the diagram above is also representative of their physical placement on the interconnect board. The number of wires that cross over each other would not work if one were to try to design a printed circuit board using this layout.

Here is a summary of the pin usage on the Arduino (Note, the names for things shown here are the names of the variables used in the software for the pin assignments):

Pin	Usage	Pin	Usage
A0	SWR_Input	A3	
A1	PWR_Input	A4	LCD_SDA (Data)
A2		A5	LCD_SCL (Clock)

D0		D7	Motor_1_Direction
D1		D8	Motor_1_Enable
D2	Encoder_Pin_B	D9	Encoder_Switch
D3	Encoder_Pin_A	D10	Tune_Button
D4	Motor_2_Direction	D11	Xmit_Key
D5	Motor_2_Step	D12	Motor_2_Enable
D6	Motor_1_Step	D13	

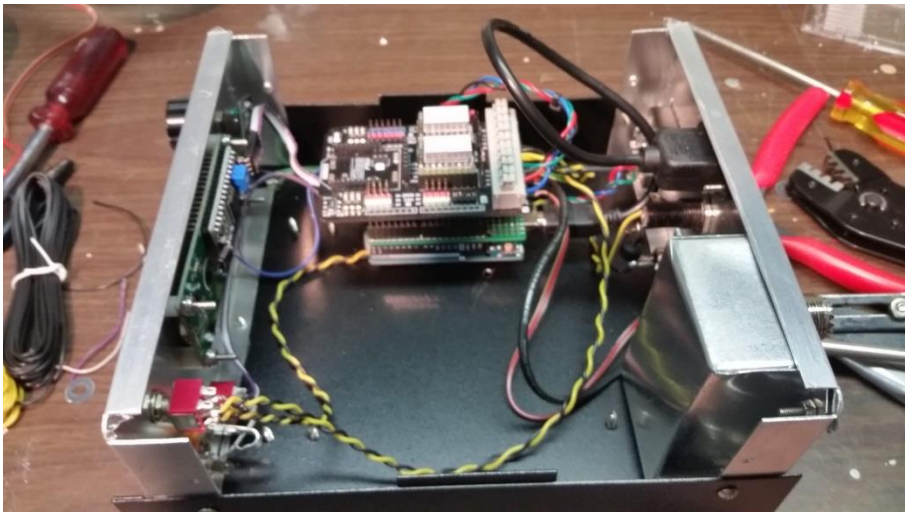
Here is a picture of the completed Interconnect Board:



Here are pictures of the almost finished controller. The enclosure is a bit oversized as far as what was needed to hold the Arduino and shields, but allowed a fairly neat front and rear panel layout. It's also the same enclosure I used for my BitX-40, so when the two are side by side, it will look reasonably professional.

Note the use of the heavy duty Cinch-Jones connector for the cable that goes to the stepper motor on the antenna. I forget what the wire gauges are in the cable, but it's an 8 conductor rotator cable that has been in place for years. You want to use fairly large conductors if the cable run is a considerable distance to prevent a large voltage drop in the cable when the motor is operating. The motor is actually designed to work on 24V, but I'm running it on 12V, so any significant loss of voltage in the cable might be too much.

I also used 10 pin CJ connectors, as I had a couple of them lying around, even though only 4 wires in the cable are currently being used. Having the extra pins and wires in the cable will allow me to do other things in the future if necessary.



One improvement I might make here is to rotate the Arduino stack 90° one way or the other. I placed it close to one side of the enclosure to maximize the space available for anything else I might want to put in the box, but it makes it really difficult to see to line up the header pins between the 3 circuit boards.

Also notice that there is a jack for a CW key on the front panel of the controller. Since the controller keys the transmitter in the auto-tune mode, I decided to put the key jack on the controller. The "Transmitter CW Key" jack on the rear panel allows a ¼" male-male jumper to connect to the transceiver. Internally, the transmitter can be keyed either directly from the Arduino's *Xmit_Key* output pin, or by the relay on the interconnect board, depending on the setting of the switch on the front panel.

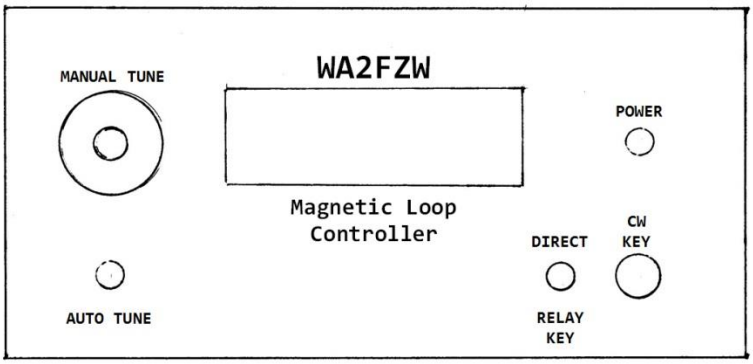
A Final Construction Note

A few of my friends that have been following and kibitzing on the project asked how I managed to do such a neat job of labeling the front and rear panels. The labeling was done using DIY decals. These come in two types. The ones used here have a clear substrate.

I used one large decal each for the front and rear panels, which is why you don't see the outlines of individual labels.

I made them using Microsoft Word. I scanned full size drawings of the front and rear panels and inserted them into a Word document. Make sure Word doesn't automatically scale the pictures, which it will if the margins on the document are set too small. Print the document and make sure it lines up with your original drawing before proceeding any further.

Once that is done, you can use Word to create textboxes to contain the label text, and you can drag them around to position them however you please. You can also use any font and color of your choosing. Here is my front panel drawing with the labels added in Word:



Once you have all the text where you want it, delete the drawing, which will leave you with a document containing just the labels (save it under a separate name), which you can now print on the decal paper. From there, follow the instructions for the particular transfer paper you are using.

Before applying the decals, it is very important to remove any burrs around the holes. Any irregularities in the surface will make it hard to apply the decal smoothly, as it will cause bubbles under the decal.

It is also important to make sure the surface is perfectly clean. I use prep-sol (available at the local auto parts store). Do not use mineral spirits or turpentine, as they leave an oily residue. The prep-sol does not leave any film. Make sure your hands are really clean, or better yet, use doctor gloves (non-powdered).

After the decal has had a few hours to dry thoroughly, trim any excess around the edges and the holes with a razor blade or X-Acto knife. I always give things a final coat of clear Krylon which I figure helps seal the edges of the decal.

The Software

The software is designed to work in conjunction with the hardware of the controller to accomplish the 4 objectives presented in that section.

The Arduino is programmed primarily using the “C” language, although the Arduino Interactive Development Environment (IDE) also supports “C++”. In the Arduino world, “C++” is primarily used in the development of standard libraries as opposed to user programs. If there were multiple things of any one particular type to deal with, I would consider creating a class (if I couldn’t find an appropriate library online). For example, if I was building a general purpose antenna tuner with 2 or 3 motors, I would most likely objectify the motors.

Note that the liquid crystal display and the rotary encoder are handled as objects.

In general, the software is pretty simple. It is also compact. There are less than 900 lines in the source file, and probably half of those are comments and blank lines added for clarity.

The Arduino has a built in bootstrap program that runs whenever it is powered up, reset or when new software is loaded into it. I assume this bootstrap program takes care of basic initialization functions for the Arduino hardware itself, and then it calls 2 user developed functions; *setup()* and *loop()*.

The *setup()* function does just what it says. The developer puts things in here that must be initialized in order for the main part of the program, i.e., the *loop()* function to perform whatever task the software is intended to do. I'm not going to address all the things that one can do with the Arduino, nor am I going to describe how to write a "C" program. There are many fine books and online resources available for that. I highly recommend "Beginning C for Arduino, Second Edition: Learn C Programming for the Arduino", by Jack Purdum as a great tutorial for both the beginner and experienced programmer.

What I will describe here, are the main functions of my controller program and give a high level description of how each of the functions works.

Initial Definitions

Before we even get to the *setup()* function itself, you will note that there are a lot of *#define* statements and other variable definitions and value assignments. These create symbolic names for any number of things used throughout the program. Any variables defined outside the scope of any particular function are *global variables*, which can be accessed and even modified by any of the functions that comprise the program. You will very rarely see constants hard coded in any software I write. Just makes maintenance that much more difficult and makes things harder to read.

The definitions and value assignments here include such things as:

- The Arduino input and output pins that are used to control the movement of the tuning motor and symbolic values for the commands to enable or disable the motors and symbolic values for the direction of rotation.
- The input pin assignments for the rotary encoder and the amount of motor rotation to be performed depending upon the state of the pushbutton switch built into the encoder.

- The pin assignments for the Auto-tune pushbutton and the pin assignments for reading the current forward and reflected power from the SWR bridge. The parameters associated with managing the timeouts for the tuning processes are also defined.
- The pin assignments and size of the LCD display are defined along with creating data buffers to be used to display information on the LCD.
- The final things done prior to the *setup()* function are to create global objects for the rotary encoder and LCD display.

Libraries

The software requires two libraries that are not part of the standard Arduino IDE.

One is for the rotary encoder and the other is for the I2C LCD display. Both are available on GitHub. Note that the LCD library used here works with the particular display that I used. If you use a different one, you might have to use a different library.

The *setup()* Function

The *setup()* function takes care of initializing all of the peripherals that make up the controller. Although we could define a lot of things outside the scope of an actual function, other than creating the encoder and LCD objects and assigning values to specific variables, prior to the *setup()* function, we couldn't actually perform the actions necessary to put the hardware components into the desired states.

This is what we do in the *setup()* function:

- Initialize the serial monitor (only used for debugging).
- Make sure the transmitter is turned off.
- Initialize the liquid crystal display.
- Set the micro-step size for the motor. By default, the motor will rotate 1.8° each time it is pulsed. This can be adjusted to smaller step sizes by switches on the motor shield, but we have to tell the software how the switches are set. I currently have them set for ¼ steps (0.45° steps), which seemed to work well during the initial and final testing.

- Set up the I/O pins for the motor. We have to tell the Arduino whether we're using the individual pins as outputs or inputs, and whether or not we need it to enable the built-in pullup resistors or not.
- Disable both motors, even though only using one. As noted earlier, having them enabled creates a lot of RF noise in the receiver.
- Set up the encoder input pins.
- Set up the pins associated with the tuning functions
- Define what to do when the encoder generates an interrupt.

The *loop()* Function

Once the *setup()* function is finished, the *loop()* function runs forever. Using a number of sub-functions, it monitors all the inputs and produces required outputs with the exception of the rotary encoder, which is handled by an interrupt mechanism.

The *loop()* function proper does the following tasks:

- Calls the *Say_Hello()* function to announce that the program is running and looking for work to do.
- Calls the *SWR_Tune()* function to see if the operator has requested the antenna to be automatically tuned.
- Calls the *Check_LCD()* function to see if anything other than the hello message needs to be displayed.

Convert_Degrees() Function

Convert_Degrees() converts the number of degrees of rotation requested into a number of steps (or partial steps, depending on the motor shield switch settings).

Because the *Step_Factor* value is a small decimal number, we will perform the math in floating point and then convert back to integer to send the number of steps to the motor.

Notice that we add 0.5 to the floating point result before converting back to an integer. This results in the step count being rounded to the nearest whole integer, as the fractional part of the answer gets truncated.

For example if *Temp* evaluated to 200.123, adding 0.5 would make it 200.623 and it would be truncated to 200. If the value of *Temp* evaluated to 200.789, adding 0.5 would make it 201.289 and it would be truncated to 201.

SWR_Tune() Function

The *SWR_Tune()* function is the heart of the controller. This function had to be almost completely rewritten between having the controller spread out all over the workbench in breadboard mode and when it was built into the final enclosure. I don't have a clue as to why the tuning algorithm had to change so drastically, but it is important to note that the values of the tuning parameters set in my version of the software might not work for anyone else who decides to build this; particularly if a different SWR bridge is used. It would also most likely require a totally different approach if a vacuum variable capacitor were to be used in lieu of the air variable capacitor.

The *SWR_Tune()* function first checks the status of the *Tune_Button*, and if it is active (LOW), we will proceed to tune the antenna based on the SWR reading. If the button hasn't been pushed, we just return to the caller.

If the button has been pushed, we turn on the transmitter and check to see if it is actually transmitting. If not, we terminate the process (Note: the *Is_Transmitting()* function takes care of turning the transmitter off and notifying the operator).

Next, we get a reading of the current SWR, and check to see if it is already at or below *Step_2_Limit*, which indicates that the antenna is already tuned for the current frequency. If this is the case, we report the SWR, kill the transmitter and return.

The actual tuning process is a two-step process.

In step 1, we use a back-and-forth search algorithm to coarse tune the antenna. We rotate the motor back and forth starting with *Step_1_Degrees* (S1D, currently set to 5°), then -2*S1D, then +3*S1D, etc. until we see the SWR drop below the value of *Step_1_Limit* (currently set to 10). Once this limit is reached, we move onto step 2.

If at any time during step 1 and step 2, the transmitter stops transmitting, or if the value of *Xmit_Timeout* (currently 5 seconds) is exceeded we will exit the tuning process and report the issue to the operator via the LCD display. We also check to see if the value of *SWR_Input* is zero. If it is, then the antenna cannot be tuned any better, so we turn off the transmitter and exit the function.

After each movement of the capacitor in both step 1 and step 2, we will compute the actual SWR reading and display it on the LCD so the operator can track the progress.

In step 2, we use the same back-and-forth search algorithm to fine tune the antenna, except now we start with *Step_2_Degrees* (S2D, currently set to 1°), then -2*S2D, then +3*S2D, etc. until we see the SWR drop below the value of *Step_2_Limit* (currently set to 1). Once this limit is reached, we are done, except for reporting the final SWR on the display.

Again, if at any time during step 2, the transmitter stops transmitting, or if the value of *Xmit_Timeout* (currently 5 seconds) is exceeded or the value of *SWR_Input* goes to zero, we will exit the tuning process and report the issue to the operator via the LCD display.

Finally, we get the final SWR reading and report it, then turn the transmitter off.

The *SWR_Tune()* function makes use of two other functions to perform the tasks of checking to see if the transmitter is still on and to check the timeout. These functions are *Is_Transmitting()* and *Check_Time()*. They are simple enough that they do not require a detailed explanation as to how they work.

Also note that in the current version of the software, a limit test was incorporated into the *Rotate_Motor()* function. This resulted in drastically shortening the amount of time required to tune the antenna. Most of the time, it now tunes in less than a second.

Nine times out of ten, the function tunes the antenna in less than a second or two, even when changing frequency from one end of the 40 meter band to the other. When it doesn’t tune it right away, it’s because the step 1 logic has actually moved the capacitor out of range.

Because I used an air variable tuning capacitor, there are 2 positions of the capacitor where the antenna will tune. If one were to use a vacuum variable, this would not be the case. I suspect that if a vacuum capacitor were to be used, this function would perhaps require a totally different approach.

One final note; in Version 1.0, the step 1 and step 2 sections of the code are almost identical. I could probably combine them into a single function, but chose not to. If I get some new idea about how to improve the tuning process, combining them might actually make such changes more difficult to implement.

The *Read_Encoder()* Function

The *Read_Encoder()* function is never called directly from any of the other functions, but rather is invoked by the Arduino's limited hardware interrupt capability. The Uno has only 2 analog pins (A2 & A3) that can be used to start an interrupt handling process. Both of these are connected to the encoder. Note that some of the more powerful Arduinos have more interrupt handling capability.

The first thing *Read_Encoder()* does is to check the state of the built-in pushbutton switch on the encoder. The switch is used to determine if the operator wants to tune the antenna in large increments *Encoder_Big_Step* (currently set to 2°), or in small steps (*Encoder_Small_Step*, currently set to 1 step).

Unlike the *SWR_Tune()* function which keys the transmitter, *Read_Encoder()* does not. We do test whether or not the transmitter is on, and if it is, we will compute the SWR and report it.

If the transmitter isn't on, we assume the operator is simply tweaking the tuning for maximum receiver noise and indicate that tuning is being done in "Receive Mode" on the display.

One thing to note; after we read the encoder there is a test to see which direction the knob is being turned. There is a *case* statement for *DIR_NONE*, followed by a *return* statement. I didn't think it would actually possible for the interrupt function to be executed if the encoder wasn't being turned, but, for some reason that I have yet to figure out, as soon as the *attachInterrupt()* function was called to set up the interrupts for both encoder pins in *setup()*, the interrupt function was being executed. Testing for no movement and simply exiting the function took care of the issue.

The *Rotate_Motor()* Function

The *Rotate_Motor()* function gets invoked from either *SWR_Tune()* or *Read_Encoder()* functions, and does the work of actually rotating the motor by some specified amount.

The function takes two integer arguments; *How_Much*, which is the number of steps the motor is to be turned, and *Limit*, which is the desired SWR reading to be achieved.

If *How_Much* is positive, we turn the motor clockwise; if *How_Much* is negative, we rotate the motor counter-clockwise.

After we enable the motor, the motor is rotated the requested number of steps by sending a series of pulses to the motor shield. The shield receives a low signal, followed by a high signal. Note that there are two *delay()* calls within the pulse loop. These are necessary to create the proper pulse widths.

After each pulse, a reading of the *SWR_Input* is taken and compared to *Limit*. If the SWR reading is less than the *Limit*, we stop turning the motor and return. I didn't particularly want to put this test in this function, but doing so reduced the tuning time to less than a second 90% of the time.

Note that when the function is called from *Read_Encoder()*, the *Limit* is set to *No_Limit*, which is set to -10, which is an impossible SWR reading. This allows the motor to rotate based solely on the movement of the encoder.

Once we're finished, we disable the motor again to prevent excessive noise in the receiver.

The *Say_Hello()* Function

The *Say_Hello()* function is responsible for displaying the message announcing that the controller is running and looking for work to do. It is called on every pass through the *loop()* function to see if the welcome message needs to be re-displayed.

Anytime anything other than the hello message is displayed (e.g. An SWR reading or error message), we set a timer. When that timer expires, we re-display the hello message.

The first thing the function does is check to see if the hello message is already on the display, and if so, it just returns.

Next, we check to see if the *Hello_Timeout* since the time some other type of message was displayed has expired. If not, then we return.

After that, it uses the *LCD_Display()* function to put the hello message on the display and sets a flag indication that the hello message is, in fact, being displayed.

The *Check_LCD()* Function

The *Check_LCD()* function is called on each pass through the *loop()* function. It looks for data in the LCD buffers, and if anything is in either one, it uses *LCD_Print()* to display them.

This approach might seem a bit strange at first, but it solved a problem caused by attempting to use the display directly from the interrupt invoked *Read_Encoder()* function. Attempting to use the display directly from within that function caused some kind of confusion in the Arduino's interrupt handling mechanism, which in turn, caused the whole program to lock up.

The problem was solved by having *Read_Encoder()* simply put messages into the buffers to be picked up in the main loop after *Read_Encoder()* had completed its work.

Note, *Check_LCD()* uses the function *Finish_LCD()* to perform the last couple of steps required to actually display the line 1 or line 2 buffer. That function is simple enough that it need not be described separately here.

The *LCD_Print()* Function

The *LCD_Print()* function handles the task of actually displaying something on the LCD display. Arguments are which line to put it on, and what to put there. Note, we allow the caller to specify lines 1 or 2, and convert those to 0 and 1 in the function (just a human factors consideration).

The function first checks to see if the requested line number actually exists on the display being used, and if not, simply ignores the display request.

We then create a standard *char[]* type buffer to transfer the information to be displayed into. We must do this, as the print functions in the LCD object do not have the capability of handling an argument of the type *String* (I may fix this somewhere down the road).

Lest we cause a problem by attempting to put too much data in the fixed buffer, we make sure the length of the data *String* we are being asked to display is no longer than the buffer space allocated (which is just enough for the width of the physical display). If it's too long, we simply truncate it.

After copying the *String* into the character buffer, we check to see if it's shorter that the display width. If that is the case, we pad the end of it with space characters. If we didn't do this, the remnants of a previous message could remain on the display.

Finally, we use the print function built into the LCD object to put the message on the screen.

The *Compute_SWR()* Function

The *Compute_SWR()* function reads the *PWR_Input* and *SWR_Input* pins and computes the actual SWR reading which it returns in a *String* object ready for displaying on the LCD.

The readings are converted to floating point, and the SWR is computed using a standard formula for the computation:

$$\Gamma = SWR\_Input / PWR\_Input$$

$$VSWR = (1 + \Gamma) / (1 - \Gamma)$$

We add 0.05 to the answer to round the answer to the nearest tenth and build a *String* object which is returned to the caller ready to be put on the LCD display.

Note that the SWR calculations seem to be fairly accurate; however I'm unable to easily compare the readings produced here to an analog meter. My little 5 watt radio doesn't seem to produce enough power to move the meter to full scale in the forward setting on either SWR meter that I have.

Display Messages

Here are examples of all the messages displayed on the LCD display:

The Finished Product

Here’s a picture of the BitX and the controller. I need to find matching knobs! But, I did get a blue display to replace the ugly yellow one on the radio, so at least those match.

Coming Attractions

There are a number of ideas I have to add functionality to the controller. As the software is only using about 35% of the available memory (both program and data) of the Arduino Uno, there is room for more code. Here are some of those ideas:

Morse Code Generator

Since the controller hardware already has the ability to key the transmitter in CW mode, with the addition of some software and a few more buttons on the front panel, it would be easy enough to give the controller the ability to send CQs and other standard messages (e.g. contest reports).

Better Motor Control

It would be nice if I can find a motor shield that provides the ability to set the micro-step setting via software control as opposed to the one I am using now, which does it by switch settings on the circuit board.

Being able to dynamically switch between settings would significantly speed up the tuning process as large rotations of the capacitor could be accomplished in fewer steps.

Transceiver Frequency Information

Using the TX/RX I/O pins on the Arduinos in both the BitX and the controller, it would seem possible for the radio to tell the controller what frequency it is tuned to.

This would allow the controller software to be able to store a table of frequencies versus capacitor settings in EPROM memory that could be used to automatically tune the antenna whenever the frequency on the transceiver is changed.

This would also require some sort of indexing mechanism on the capacitor, but since there are lots of extra wires in the control cable, this shouldn’t be too difficult.

Some Final Thoughts

Unless someone builds an antenna exactly like mine and places it in an environment exactly the same as where mine is placed, chances are that the software I have provided is not going to work without some modifications, particularly in the *SWR_Tune()* function and the associated tuning parameters.

As mentioned earlier, that code had to be completely rewritten when I moved the controller hardware from a jumble of wires and breadboards on the workbench to the final configuration. I can’t say exactly why that was, but I can guess that at least one of the factors is probably that the SWR bridge board behaved somewhat differently in the original PDC1 enclosure than it did in my enclosure.

Concentrating the components and wiring on the Interconnect Board may have also induced some kinds of interactions between the various signals as well. Who knows?

The point is that this project can be used as a model for your own creation. As I mentioned in the introduction, although I found a number of designs for antennas and controllers on the internet, I wanted to take a crack at designing and building my own. Call me a glutton for punishment if you will, but I like such a challenge!

If you decide to build something based on my work, please feel free to contact me via any of the websites I have published this on with questions or ideas for improvement. I can also be contacted at WA2FZW@ARRL.net. Good luck!