A Smart Controller for the AR-22 Rotator
Version 05.2

© John Price – WA2FZW

License Information

Introduction

I had just finished up two Arduino based ham radio projects; a BitX-40 QRP transceiver and a magnetic loop antenna and controller when I received an email from Mike, WV2ZOW asking if it might be possible to design a computer based controller for the old CDE AR-22 (clunk-clunk) rotator. Without even a second thought, I replied something to the effect of “Piece of cake”! After all, I had spent a number of years at the phone factory (AT&T) computerizing things that were never designed to be computerized, so this seemed to be a project right up my alley.

The overall objective was to be able to control the rotator either manually or with the N1MM+ contest logging and rotator control software.

I looked at a number of similar projects online and one in the ARRL’s *Arduino for Ham Radio* book and although I got a few good ideas, didn’t find any that really looked like they would accomplish what I wanted to do. My main issue with most of them was the software. I looked at one controller that has some 14K lines of code. My software has about 1,400 lines and about half of those are comments and blank lines inserted for clarity.

Well, it turned out not to be quite as simple as I first thought (definitely an understatement), but overall, it wasn’t totally impossible.

Thanks to Mike (WV2ZOW) for all his help with the electrical issues encountered along the way.

Here, I'll describe the hardware and software used in the project. The reader can build this exactly as described or use this description as a model for a different approach. I'll try to point out all the wrong assumptions and mistakes I made along the way.

Revision History

Changes in Version 2.0

In version 1.0, the azimuth was completely controlled by pulses generated by the cam operated switch inside the rotator itself. Those pulses were sent approximately every 6° of rotation.

In Version 2.0, interrupts from a software timer were used to generate pseudo index pulses approximately every 1° of rotation, which allowed the azimuth to be set within 1° (plus or minus a slight error).

Changes in Version 2.1

Version 2.1 included three hardware changes and one software change.

The first change which included both hardware and software was done to fix the problem of the Arduino rebooting whenever the N1MM+ Rotor program or the Arduino IDE serial monitor was started. A toggle switch labeled "Run/Update" was added to the back panel. This switch, when closed, places a 10uF capacitor between the Arduino's reset pin and ground. It gobbles up the very brief reset pulse sent by the serial connection whenever the N1MM+ Rotor program or the serial monitor is started.

The reset pulse is generated via a connection on the Arduino board between the DTR (Data Terminal Ready) line from the serial interface to the reset line.

That same short reset pulse is also sent from the Arduino IDE whenever we need to load new software into the Arduino, thus the switch must be opened to disconnect the capacitor whenever new software needs to be loaded.

Associated with this modification, the back panel “Reset” button was disconnected from the Arduino’s reset pin and moved to D19 (an interrupt enabled pin). Pressing the button will now do two things; first, it will save the *currentAzimuth* then perform a software initiated reboot.

The second minor hardware change was to add diode D2 (I renumbered them on the schematic also) between the output of the LM7812 regulator and the Arduino’s Vin pin. The connection for the Power indicator LED is now between the regulator and D2.

This was done because in Version 1.0, the LED would light when the controller was connected to a computer via the USB connection regardless of whether the internal power supply was turned on or not. This was because when the Arduino was connected to the computer via the USB port, there was approximately 4.5V on the Vin pin; that was totally unexpected. Now, the Power LED only lights when the internal power supply is on. When the USB connection is present, the LCD display will continue to light up regardless of the state of the internal supply.

Note that the Version 2.1 or Version 3.0 software will run on controllers that do not have these hardware modifications.

A third hardware modification was to build the power supply on a separate circuit board that mounts on top of the interconnect board. The circuitry was not changed (other than the addition of ‘D2’); this modification just made a little more total real estate available and made it simpler to wire it all up.

Changes in Version 2.2

In version 2.2, two new functions were added, and the timing parameters were tweaked.

The first new function is *Backup()*, which moves the rotator slightly off the end stop whenever it times out (either in normal use or calibration). The second is *Ck_Reboot()*, which checks to see if the operator pushed the reset button on the back panel. It contains the same logic that was originally coded in the *Loop()* function, but since it also needed to be checked in the *Calibrate()* function, it was converted into a stand-alone function.

In conjunction with the addition of the *Backup()* function a new definition was added called *CAL_ADJUSTMENT*, which defines the number of degrees to back off the end stop.

The timing tweaks included reducing the timer interrupts from 141mS to 140mS and changing the rotator timeout from 1 second to 850mS.

Changes in Version 3.0

After adding the timer interrupt in Version 2.0, I noticed that the rotator was not turning a full 360° when commanded to do so (either via the serial interface or the encoder). Hooking the rotator up to the original clunk-clunk box, I also discovered that the same thing was happening; even though the original controller recorded 60 clunks, the rotator did not turn a full 360°.

Making some careful measurements, I found that it was coming up about 9° short. Doing a little math, I calculated that the cam switch was actually producing pulses about every 5.85° instead of the advertised 6°.

This necessitated a complete overhaul of the azimuth computations.

Rather than incrementing or decrementing the *currentAzimuth* by a whole degree for every timer or cam pulse received, the azimuth is adjusted by 0.975 degrees (5.85/6) for each pulse. Rather than attempting to use floating point math, all the internal azimuths and adjustment factors are multiplied by 1000 internally and the *Long* data type is used to store the numbers.

Also in Version 3.0, because of all the things that might need to be tweaked for a particular rotator, I moved the definition of the number of degrees per cam pulse and all the variables associated with the timer interrupt interval, cam pulse detection, LCD parameters and encoder pin definitions to the *My_Rotator.h* header file to make it easier for someone to find and tweak these parameters.

Changes in Version 4.0

Although I finished the initial version of this project some 6 months ago, I hadn't actually used the controller in a real operating environment until I finally got a 6 meter beam on the roof just before the 2018 ARRL June VHF contest. In operating the contest, the first thing I discovered was that it took an awful lot of turns of the azimuth (encoder) knob to turn the antenna 180°. The second thing I noticed was that in the brief times when I was repositioning my fingers on the knob to turn it more, sometimes the azimuth would be saved before I had finished setting the final target azimuth.

So, in Version 4.0, I made 2 hardware/software changes. First, I added a 3 position SPDT (center off) toggle switch to the front panel. This switch along with some software changes allows the operator to select the number of degrees to increment (or decrement) the target azimuth when turning the encoder knob. The increment can now be set to 1°, 5°, or 10°. A symbol definition (*AZ_SW_INSTALLED*) was added to the *My_Rotator.h* file to indicate whether or not said switch is actually installed.

If the switch is not installed, the azimuth increment is determined by the value set in the *My_Rotator.h* file for *DEFAULT_INCR*; it is set to 5° in the distributed software but can be changed to any value you like.

A second hardware change I made was to add another LED to the front panel which will indicate that the *currentAzimuth* has been saved in the EEPROM.

A second software change was to modify the algorithm for saving the azimuth. As noted above, I noticed that it would sometimes get saved before I had finished setting a new target azimuth. Now the azimuth will be saved 10 seconds from the last time the rotator stopped turning. This can be adjusted by changing the value of the symbol *EEPROM_TIMEOUT* in the "My_Rotator.h" file.

Changes in Version 4.1

In Version 4.1, I made a slight change to the calibration logic. In previous versions, when a timeout occurred due to missing a cam pulse, the *currentAzimuth* would get set to either 0° or 360° based on which direction the rotator was turning. It the timeout did not occur due to it hitting the end stop, it would now be way out of calibration.

Doing a calibration after such a timeout would often cause the rotator to turn through almost a full rotation to re-calibrate itself.

In order to correct this, a new variable, *lastGoodAzimuth* is introduced. This gets set to the *currentAzimuth* only after successfully seeing a sync pulse (hard or soft). In the calibration function, this variable is used to determine which way to turn the rotator to do the calibration instead of the possibly erroneous *currentAzimuth*.

Changes in Version 5.0 & 5.1

There are no significant software changes in Version 5.0 other than to reassign some of the Arduino pin numbers used.

What did change is the hardware. Using my newly acquired knowledge of how to design printed circuit boards, I decided to replace the hand-wired boards used in the first two of these I built. That also allowed me to fix a hardware bug in the first one that was just too hard to fix because of the way I built the hand-wired board for that one.

Version 5.1 fixed some bugs on the PCB and in the software.

Changes in Version 5.2

When testing Version 5.1 with a 2nd rotator, a problem showed up in that the rotator was hitting the mechanical end stop with the cam switch in the closed position, which caused a never-ending pulse to be sent to the controller and thus, the expected timeout at the end stop would never occur.

I made a number of attempts to fix this in the software without any luck and finally decided to re-design the entire pulse detection circuit. This also required a couple more minor tweaks in the code.

Cleaning up the pulse detection hardware and software also allowed me to further tweak the timing parameters.

The N1MM+ Rotator Control Program

The N1MM+ Rotator Control program supports a number of different types of rotators and communications protocols. Two of those protocols seem to be fairly standard. Those are Yaesu's GS-232 protocol and HyGain's DCU-1 protocol.

I originally thought it might be a good idea to design the AR-22 controller so that I could support either protocol, which might allow it to be used with other contest logging programs, but in the end, decided that the GS-232 protocol was much easier to implement and it seemed to have more capabilities than the DCU-1 protocol, including support for 2-axis systems (not implemented in the AR-22 software).

The N1MM+ program doesn't even use the full basic GS-232 protocol. The only commands implemented seem to be:

Mnnn	Turn the rotator to the azimuth specified by nnn.
C	Ask the rotator what its current azimuth is.
S	Stop the current command.

The N1MM+ Rotator Control program can be used as a stand-alone controller, or it can be used in conjunction with the N1MM+ Contest Logging program, which determines an approximate antenna azimuth based on the coordinates of the station using the software and the station being communicated with's call sign.

Since the AR-22 software emulates the Yaesu protocol, when setting the N1MM+ program up, you have to tell N1MM+ that the AR-22 is a Yaesu type rotator. You can also give it a unique name.

The N1MM+ program communicates with the AR-22 controller using a USB-2.0 port running at 9600 baud (can't be changed in the N1MM+ Rotor program). The N1MM+ Logging software also supports the ability to network a number of computers together for multi-operator contest operations.

Other Rotator Control Possibilities

Since the AR-22 controller software implements all of the basic Yaesu rotator commands, it will probably work with any PC based rotator control software which supports the GS-232 protocol.

Although I’m not doing any EME or satellite work, there are programs out there that are capable of tracking the moon or man-made satellites and feeding control information to the Yaesu rotators. Since the software for the controller implements the complete Yaesu’s GS-232 protocol any of these programs should work.

Note that you can also control the rotator by simply using the serial monitor window in the Arduino IDE; just be sure to set it up to use a carriage return as the end-of-line character.

For anyone proficient at writing code for the PC or maybe a Raspberry Pi, you could also write your own program. If anyone does so, please let me know!

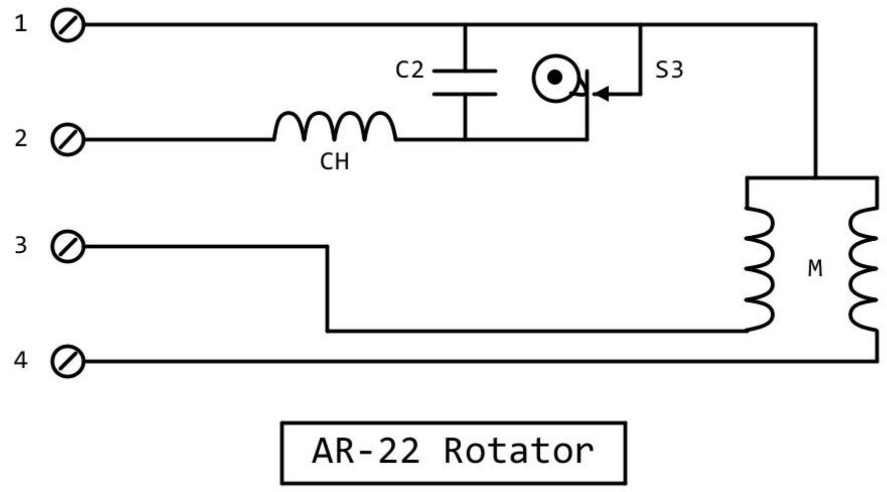
The Controller Hardware

There are only a few major components in the controller hardware; specifically:

- The Arduino Mega2560 R3 processor.
- An I2C 1602 2 line by 16 character LCD display.
- A Keyes KY-040 Rotary Encoder
- A power supply to provide AC for the rotator motor and DC to power the Arduino and its peripherals.
- A custom shield containing the motor control relays and a circuit to convert the *MOTOR_INDEX* pulse from AC to DC plus the interconnections between the peripherals and the Arduino’s I/O pins.
- A custom shield containing the power supply circuitry.

The Rotator Itself

Here is the schematic for the electrical components of the AR-22 rotator itself:



‘C2’ is 0.22uF @ 400V, and ‘CH’ is 15 turns of wire 3/8” in diameter.

In the original controller box, terminal ‘1’ is connected to one side of a (not center) tapped transformer, and terminal ‘2’ is connected to the other side of the transformer through the clunk-clunk solenoid.

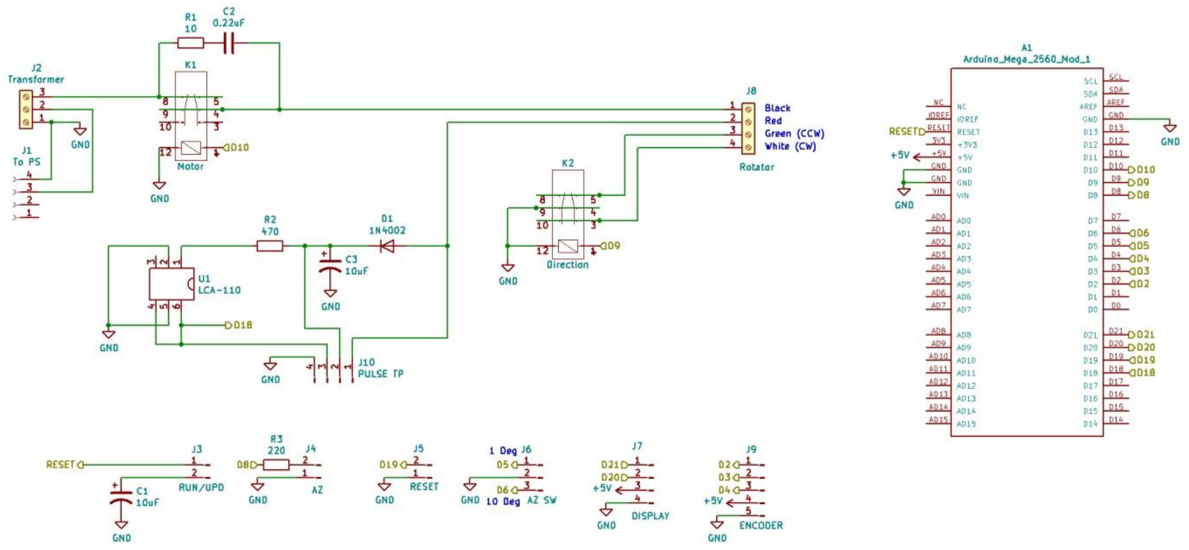
Grounding terminal ‘3’ or ‘4’ starts the motor rotating in one direction or the other.

‘S3’ is operated by a cam and sends a pulse to the control box approximately every 6° of rotation.

The transformer used in the original controller is a custom transformer that CDE must have designed specifically for the controller (their main business was always capacitors and transformers). The original transformer provided about 36V on one side of the tap (connected to terminal ‘1’ on the rotator) when the motor wasn’t running, and 6.3V on the other side of the tap (for the #47 light bulb). The motor voltage is about 28V when the motor is operating.

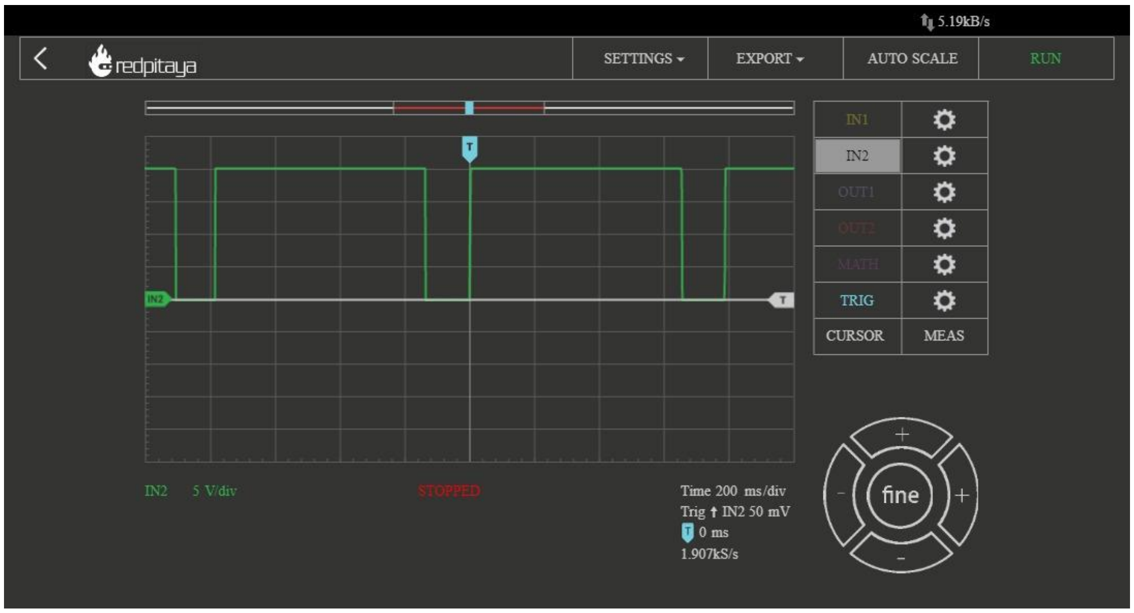
The Smart Controller

Here is the schematic for the controller shield:



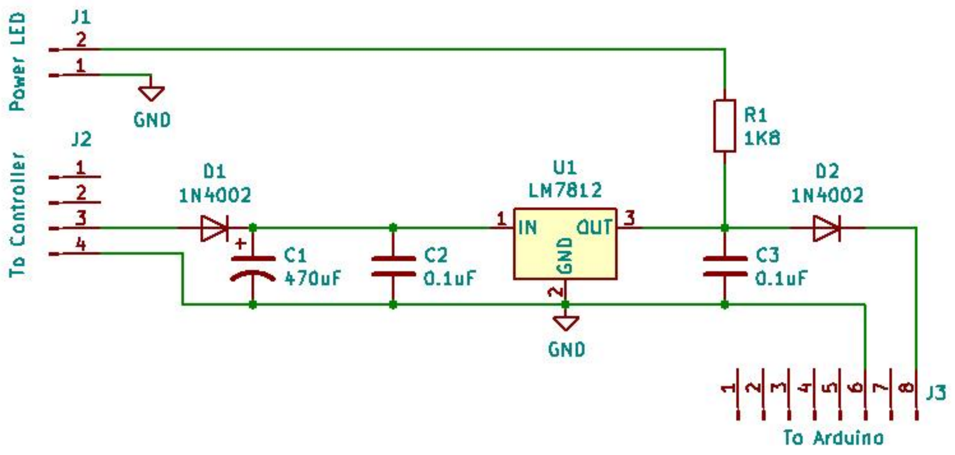
Not shown in the schematic are the power plug, power switch and 2A fuse and the power transformer (28V – CT; Hammond model 187E28).

The main change in Version 5.2 was to replace the reed relay formerly used to detect the sync pulse with a solid state relay. This resulted in the processer seeing a much better defined and cleaner index pulse as shown here:



I'm not sure why the above scope trace shows 5V/division as the vertical scale; the Red Piyata oscilloscope app has some issues! The pulse is actually 5V in height.

Here is the schematic for the power supply/regulator board that sits atop the main controller board:



In the original clunk-clunk box, power was only applied to the power transformer when the dial was turned to start the rotator moving. Here, I not only use the transformer to power the rotator motor, but also as part of the DC power supply for the Arduino and its associated components, so it will always be turned on.

The original controller included a thermal protection switch on the primary of the power transformer; the smart controller omits it. In years of using the AR-22, I never had it activate. I believe it primarily came into play in the case where the rotator hit the mechanical end stop and the operator did not manually move the solenoid lever on the bottom of the box to calibrate the controller within a reasonable amount of time.

I determined that the thermal switch was unnecessary in the smart controller as the transformer used seems more robust than the original transformer and the software shuts down the motor automatically whenever the end stop is hit.

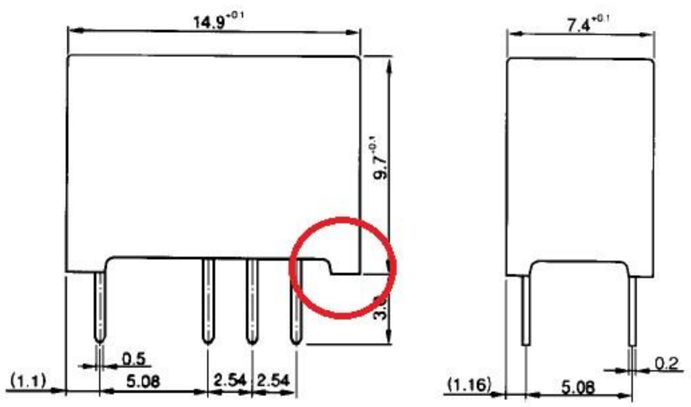
The top half of the transformer secondary provides 28 volts to the motor through ‘K1’, and grounding the other end of one or the other of the motor coils via ‘K2’ determines the direction of rotation. In the original clunk-clunk box, that was done by one of the cleverest mechanical switches I’ve ever seen that was part of the dial mechanism.

Here, the software will set the direction of rotation by activating (or not activating) ‘K2’. After a slight delay, ‘K1’ will be then be activated to start the rotation.

The relays I used for ‘K1’ and ‘K2’ have 5V coils and contacts rated for 2A. The rotator motor draws just a bit over 2A, so since the relays are actually DPDT relays, I’m using both sets of contacts just to be on the conservative side. The relay coils draw 30mA, so they can be operated directly from the Arduino I/O pins, which are rated for a maximum of 40mA.

Note the combination of a 10Ω resistor and 0.22uF capacitor across the ‘K1’ contacts. This RC circuit prevents arcing of the ‘K1’ relay contacts when the motor is turned off. There is no such protection on the direction relay (‘K2’) as that one is never operated by the software while power is being supplied to the rotator through ‘K1’.

Something to note about these relays; they have little bumps on the bottom side corners as shown here:



If you’re going to solder them into the board, it’s not a problem, but they don’t socket well. I used a pair of flat cutters to cut the bumps off, but be careful on the end where the pins are close to the corner.

Terminal ‘2’ on the rotator is used to send pulses back to the controller via a cam operated switch that is closed momentarily every time the rotator moves about 6°. In Version 5.2, the entire pulse detection circuit was modified to use a solid state relay in place of the reed relay formerly used.

‘C3’ is the motor starting capacitor (not shown on the schematic; it is connected between terminals 3 and 4 of the rotator connection) and has a value of approximately 140uF and an operating voltage of 165VAC.

Summary of Arduino input/output pin usage

These are the pin assignments for the Arduino Mega 2560:

- D0 USB Interface Receive
- D1 USB Interface Transmit
- D2 ENCODER_PIN_B
- D3 ENCODER_PIN_A
- D4 CAL_SWITCH (encoder pushbutton)
- D5 ONE_DEGREE (1 degree switch)
- D6 TEN_DEGREE (10 degree switch)
- D8 SAVED_LED (Azimuth saved LED)
- D9 MOTOR_DIRECTION (Relay ‘K2’)
- D10 MOTOR_RUN (Relay ‘K1’)
- D11 Reserved for Timer1 PWM output signals
- D12 Reserved for Timer1 PWM output signals
- D13 Reserved for Timer1 PWM output signals
- D18 MOTOR_INDEX (Interrupt enabled pin)
- D19 SOFT_RESET_PIN (Interrupt enabled pin)
- D20 LCD_SDA (I2C Data Line)
- D21 LCD_SCL (I2C Clock Line)

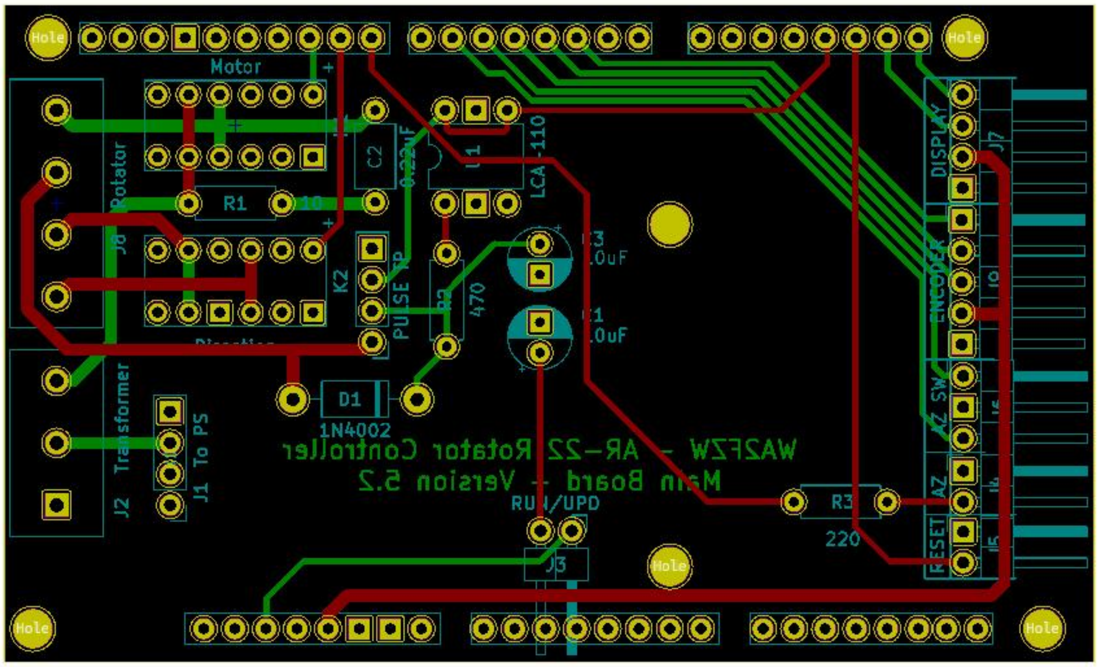
Note that some encoders found online are wired backwards. If you have that issue, simply flip-flop the pin definitions for pins ‘A’ and ‘B’.

Construction

The Controller Board

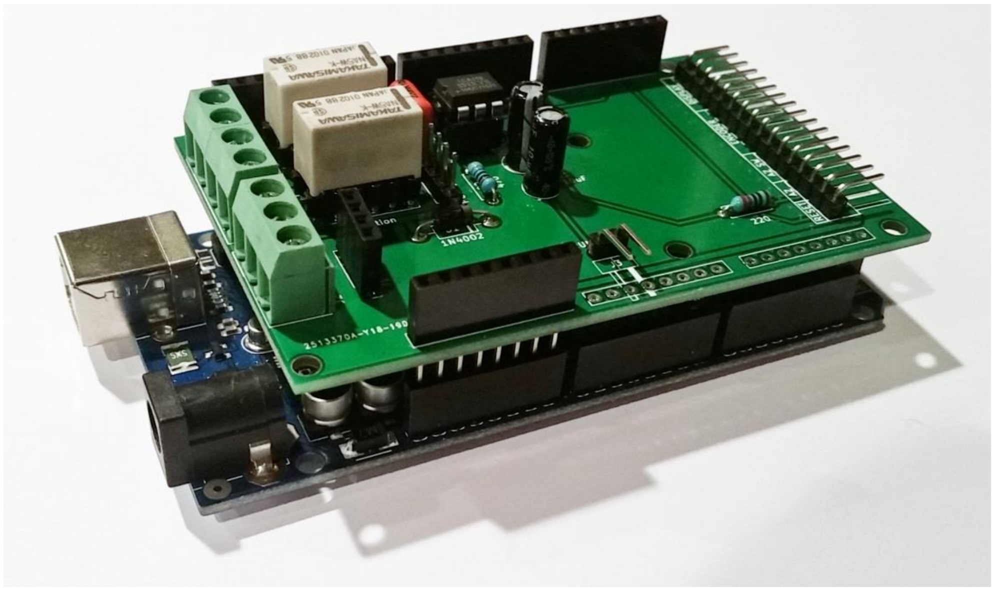
The relays and headers to connect the Arduino to the LCD, rotary encoder and a few other things are all mounted on the main controller board.

Here's the PCB layout; red traces are on the component side of the board and green traces or on the back side (X-Ray view). The ground fill areas are not shown:



Besides changing the reed relay to an SSR in Version 5.2, I also added the "PULSE TP" header to the board to make it convenient to use an oscilloscope to look at the raw pulse from the rotator, the rectified and filtered pulse and the signal going to pin D18 of the Arduino. I didn't have room to label which pin is which, but it's easy enough to figure out by looking at the PCB layout above.

Here’s what the controller board looks like mounted atop the Arduino Mega:



The right angle male headers on the right side of the picture are the connections for the LCD, encoder, reset button and the Run/Update switch.

The right angle connector on the side of the board toward the camera is the connection to the “Run/Update” switch on the rear panel. There wasn’t room to put it with the rest of the connectors. A word about its purpose in life; the Arduino IDE sends a brief DTR pulse when it wants to update the software. This pulse also gets routed to the “Reset” pin of the Arduino. If the Arduino doesn’t see new software in a couple of seconds, it simply reboots. Unfortunately, N1MM+ sends the same pulse when it starts, which causes the processor to reboot. The switch puts a 10uF capacitor across the “Reset” pin when it is in the “Run” position. The capacitor gobbles up the DTR pulse thus preventing a reboot. To load new software, the switch must be changed to the “Update” position.

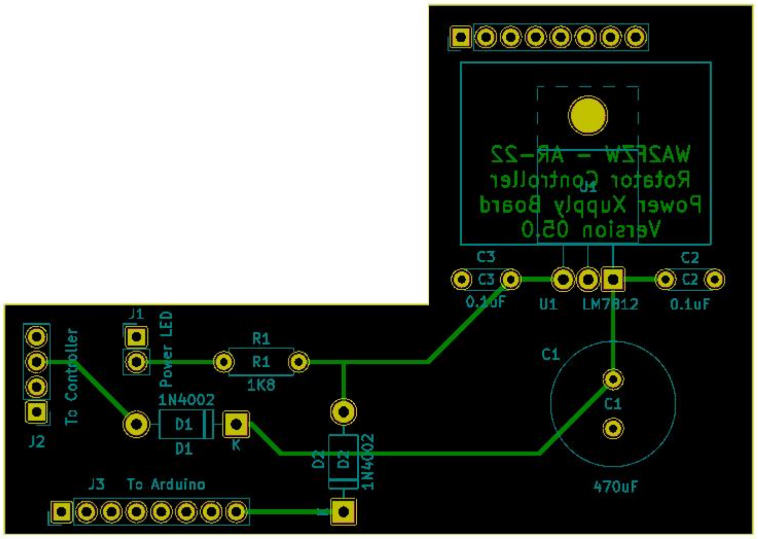
The screw terminal headers on the left are for the rotator and the transformer, where we want to use a heavier gauge wire for the connections.

The four-pin female header just behind the screw-terminal headers provides a connection between the ground and center tap leads from ‘T1’ to the power supply board.

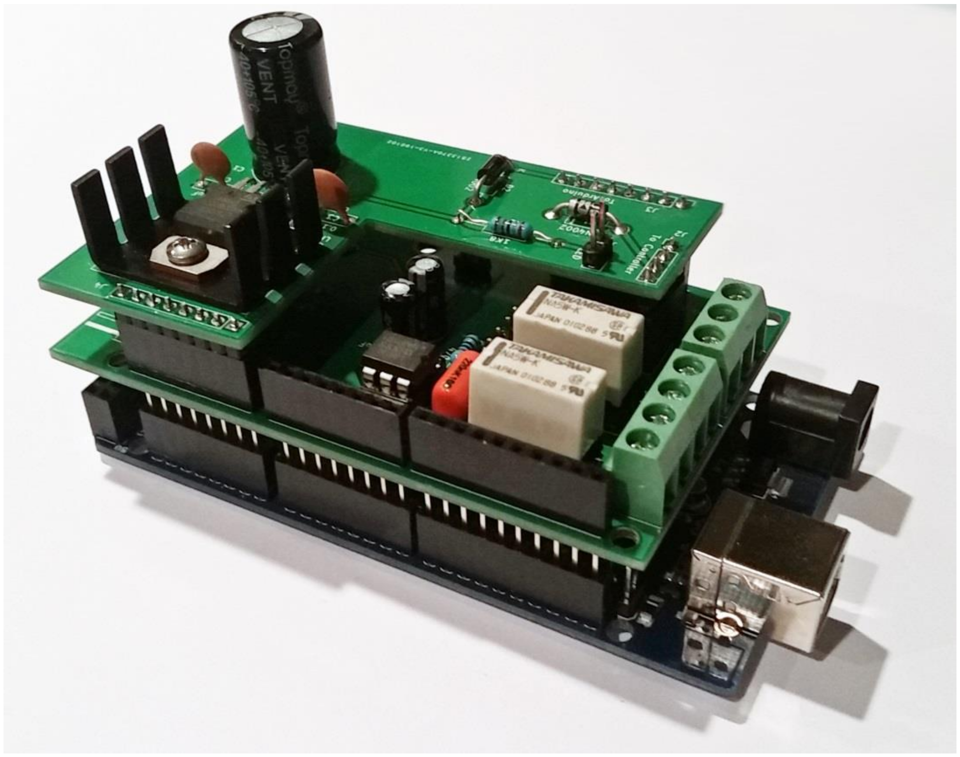
Power Supply Board

To provide a bit more real estate on the interconnect board, in Version 2.1, the power supply components and connection to the power LED were moved to a separate circuit board that mounts atop the interconnect board. This also allowed more room a better heat sink for the regulator. Note the regulator does get warm!

Here’s the PCB layout for the power supply board; note, all the traces on this one are on the back side of the board (X-Ray view):



Here's what the power supply board looks like mounted on top of the stack. The vertical pin header is the connection for the power LED:



Metal Work

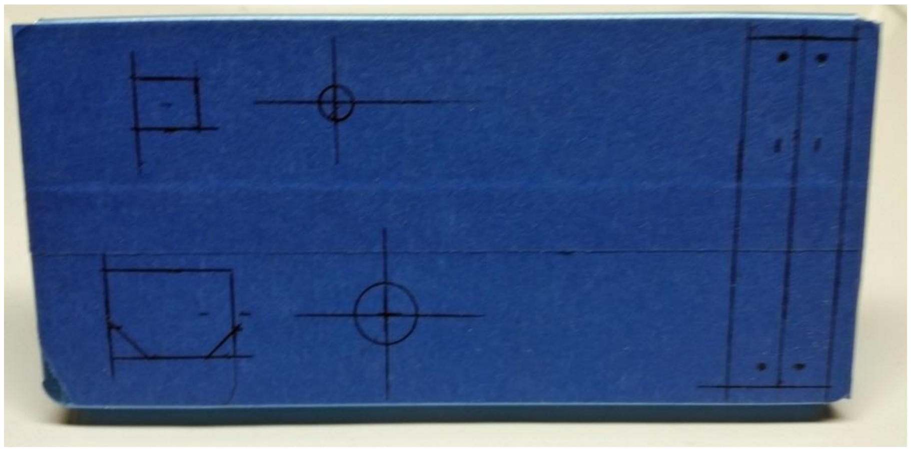
The enclosure I used is from Circuit Specialists. They have a good assortment of enclosures for reasonable prices. I've used this same enclosure on two other projects already.

I've seen a lot of nice projects where the builders have had issues with doing a good job of cutting holes; particularly ones that aren't round.

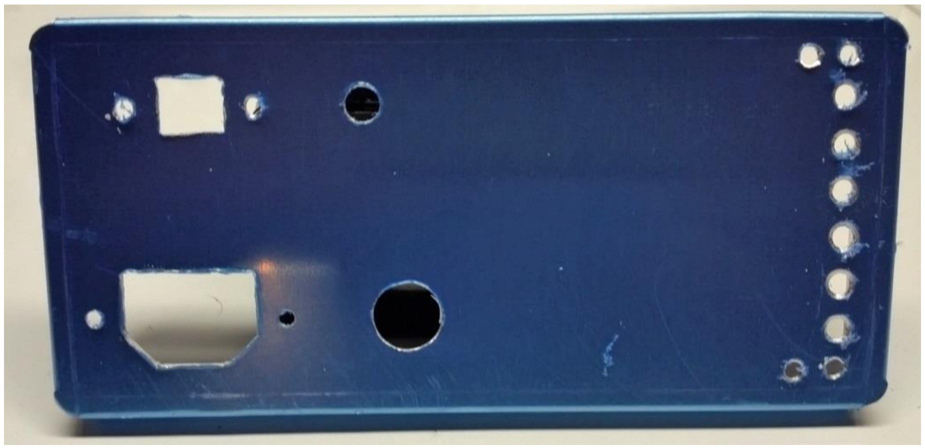
A couple of tools that I find to be an absolute necessity are a metal nibbler; mine is an Adel Nibbler (it's a bit pricey, but worth it), and a step drill (I have 2 different sizes). Again, not the cheapest tool, but it makes much nicer holes (particularly big ones) than regular drill bits. Regular drill bits make holes that are slightly triangular according to my high school metal shop teacher.

The first step in doing nice metal work is to make a good drawing of what you want to do, as shown below in the section titled “Labeling”. Then, you need to transfer that drawing to the metal. I cover the metal with masking tape and then using my “blueprint” re-draw the plan on the masking tape. Note the masking tape also keeps you from scratching the metal (unless you really slip up). Use a square to make sure all the lines are parallel to the edges of the panel or box.

Here’s the (Version 1.0; no “Run/Update” switch or “Reset” button) back panel ready to be cut and drilled:



And here’s what it looks like after drilling and cutting (it’s not really blue; that’s a protective plastic coating that comes on the panel):



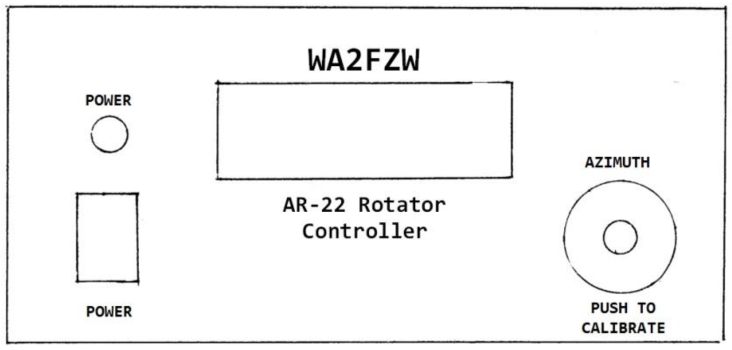
Labeling

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. You can also get them with a white substrate if you want white lettering on a colored background.

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.

I always print the decals on plain paper (or drafting vellum) first then lay that over the panel just to make sure everything is where I want it.

When you print the decal sheet, it is also important to set the printer options for “Glossy Photo Paper” and “Fine” or “High Quality” printing. I tried using “Best Quality” printing, but found that my printer puts way too much ink on the decal in that mode which makes it too easy to smear. Whichever you use, let the ink dry for a couple of hours before doing anything else.

The decal instructions tell you to spray the decal with a number of coats of clear acrylic spray; I use Krylon crystal clear although I guess you could also use a satin finish.

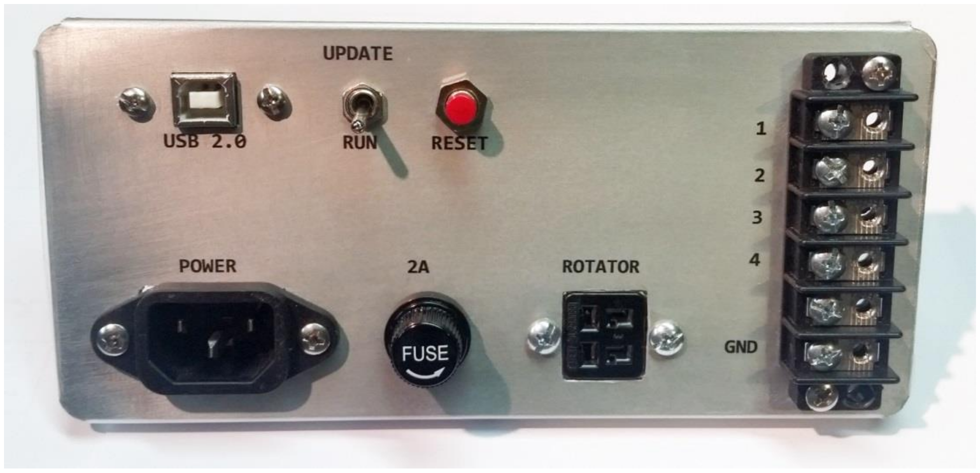
Before applying the decals, be sure to remove any burrs around the holes. Any irregularities in the surface will make it hard to apply the decal smoothly, as it will cause bubbles under the decal.

It is also important to make sure the surface is perfectly clean. I initially use prep-sol (available at the local auto parts store), sometimes followed by MEK. 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). If you want to use the MEK on a painted surface, test it first to make sure that it’s not going to remove the paint.

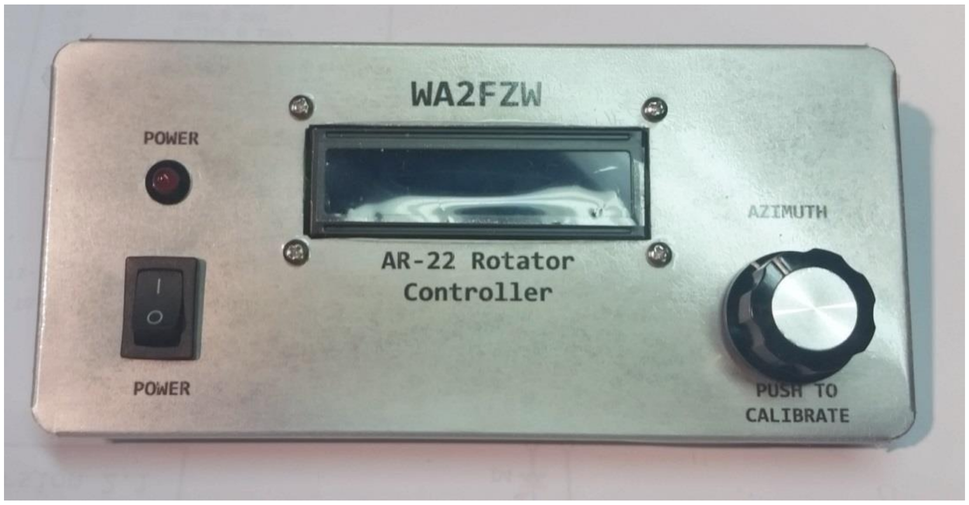
Once you have the decal lined up on the panel, use a sharp razor blade or X-Acto knife cut “Xs” in the big holes (like the one for the display). This seems to make it easier to smooth the bubbles out of the decal.

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. You can also use a fine file or emery board. I always give things a final coat of clear Krylon which I figure helps seal the edges of the decal.

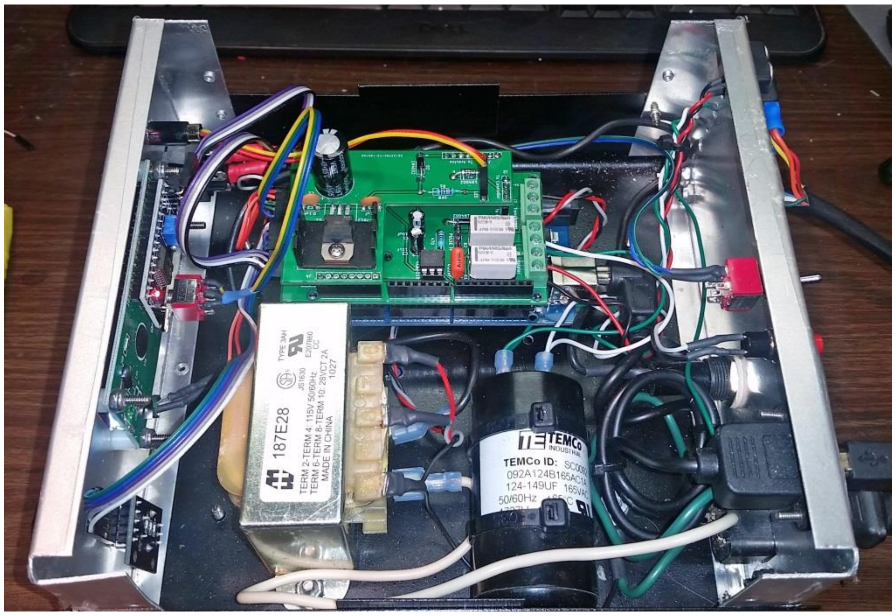
And here’s the completed rear panel:

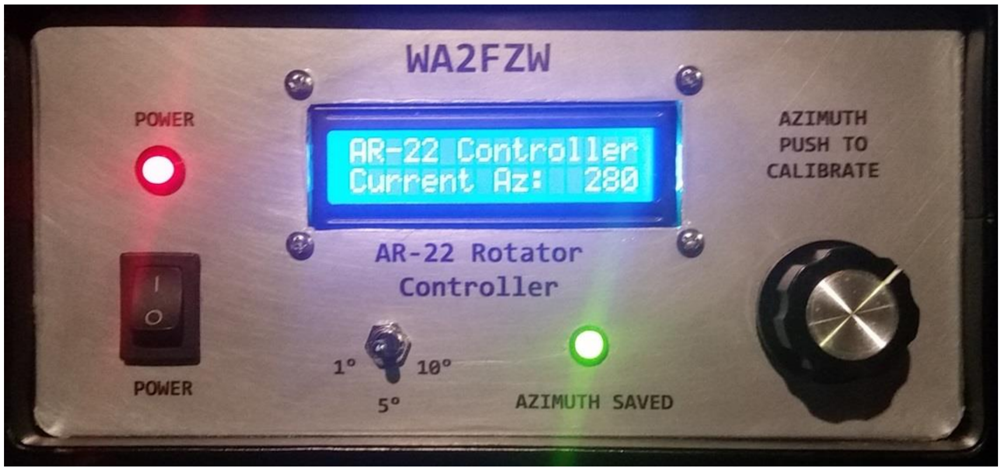


And the front panel (pre Version 4.0 – No azimuth increment select switch or azimuth saved LED):



And here’s the finished controller:





This picture shows the additional switch and LED added in Version 4.0.

The AR-22 Controller Software

I chose the Arduino Mega2560 R3 (I actually used a non-Arduino clone) as the processor, because of the increased interrupt handling capability of that version over other versions of the Arduino.

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.

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

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 two 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 the AR-22 controller program and give a high level description of how each of the functions works.

General Logic

Before starting to describe the software in any level of detail, there are a few things the reader should understand.

First of all, the AR-22 rotator itself has a cam operated switch that sends a pulse to the controller (our smart one or the old clunk-clunk box) that indicates it has moved about 6° in either direction. In Version 1.0 of the software, the azimuth resolution was controlled solely by pulses from the cam switch. In Version 2.0, a timer interrupt mechanism was added to provide pseudo index pulses approximately every 1° of rotation, thus allowing the rotator to be aimed more precisely. While that might be of little importance if the antenna being turned is a tribander or some other fairly broad-pattern antenna, it does become important if the rotator is being used to swing a many-element VHF or UHF array.

Note, as mentioned above, the amount of contact bounce in the cam switch is horrible, and in the early versions of the controller software, there was some rather convoluted software to deal with the problem. The hardware and software modifications in Version 5.2 pretty much eliminated the contact bounce problem and some additional modifications were made to the software.

The AR-22 can get out of calibration quite easily. The most common way this happens is when a movement of the rotator is followed by one in the opposite direction. When the index pulse that indicates the *targetAzimuth* has been reached is received, the rotation stops, but not immediately. Unlike some of the heavy duty rotators, the AR-22 has no brake, so even though the motor is turned off, rotation will continue because of the rotational inertia of whatever it is turning (the bigger the antenna, the more inertia).

If it stopped after receiving a real index pulse from the cam switch (as opposed to a pseudo pulse from the timer) it will have turned about 0.6° past the switch closure due to the debounce masking algorithm used. Upon starting in the opposite direction, it will receive another real pulse almost immediately.

The good news is that because of the 1° resolution in Version 2.0 of the software, the rotator will now be only 1° out of calibration as opposed to 6° as was the case in Version 1.0 of the software, and then, only if it stopped because of a hard pulse as opposed to a timer generated one.

Note that this software will probably also work with the old Alliance clunk-clunk rotators if the values of the timing parameters are adjusted for that rotator (more about this later).

The hardware would, however, have to be modified. The Alliance motor runs on 18 volts, so ‘T1’ would have to be changed. Since the center tap feeding the rectifier would then be providing only about 9 volts, the power supply circuity might also need to change.

Initial Definitions

Definitions in *My_Rotator.h*

In Version 3.0, I moved all the variables that might need to be tweaked for a particular rotator or the use of different components in the controller into the *My_Rotator.h* header file along with detailed explanations of why and how they might need to be modified.

The variables defined in *My_Rotator.h* are:

- *PULSE_DEGREES*; the number of degrees that the rotator turns for each pulse from the cam operated switch multiplied by 1000. Note that in the distributed software, this is set to 5.85 degrees, which is what I measured on the test rotator.
- *AZ_ROUNDING*; since the internal azimuths are no longer nice even multiples of 1°, we need to round them off before displaying them.
- *timerInterval*; in the distributed software is set to 137 mS. Again, this was based on the rotators I used for testing and how it is determined is described further on in this document.
- *CAL_ADJUSTMENT*; One of the things I discovered in testing is that there are actually more than 360° between the end stops on the rotator. *CAL_ADJUSTMENT* is the number of degrees to back up when the end stop is hit.
- *PULSE_DEBOUNCE*; the number of milliseconds needed to mask the contact bounce in the cam switch.
- *ROTATOR_TIMEOUT*; The number of milliseconds that are allowed to elapse without seeing a cam switch pulse indicating that the rotator hit the mechanical end stop.
- The parameters associated with the LCD display. If a different display is used, the size parameters need to change, and the I2C buss address might need to be changed.
- The pin assignments for the rotary encoder. Some have the ‘A’ and ‘B’ pins reversed. If yours works backwards, just reverse the pin assignments.

- *AZ_SW_INSTALLED*; If the 3 position toggle switch to set the target azimuth increment is installed, the definition of this symbol should be enabled. If the switch is not installed, comment the line out.
- *LCD_ADDRESS*; The I2C address of the display. The standard I2C address for the LCD is 0x27. One of my units has one whose address is 0x3F. There are programs available to scan for I2C devices and report their address available on line if you're not sure. If the display lights up, but doesn't contain any information it's a good guess that the address is wrong.
- *DEFAULT_INCR*; If the toggle switch is not installed, this defines the number of degrees to increment (or decrement) the target azimuth for each click of the encoder. It is set to 5° in the software as distributed, and may be changed to whatever value you like.
- *EEPROM_TIMEOUT*; The number of milliseconds to wait after the rotator has stopped turning to update the EEPROM.

Definitions in the Main Program

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.

The definitions and value assignments here include:

- Timeout parameters for the rotator itself, the display of the default "Hello" message and the EEPROM update.
- Definitions of the standard messages that are displayed.
- The definitions of the numerical values associated with each of the alphabetic commands used between the N1MM+ software and the AR-22 controller software.
- The definitions of the alphabetic commands used between the two programs. Note, the entire basic set of GS-232 commands is included in the list even though not all are used by the N1MM+ program.

- Setup the *commandTable*, which provides the cross reference between the alphabetic commands and the numerical values used within the AR-22 program. Note that I could have simply used only the alphabetic commands, but should I ever decide to support the DUC-1 (or any other) protocol, this approach will make it simpler to do that.
- The final things done prior to the *setup()* function are to create global objects for the rotary encoder and LCD display.

Arduino Libraries

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

One is for the rotary encoder. The second is for the I2C LCD display, the third is for the timer, and the fourth for the software reset capability. All four are available on GitHub. Note that there are a number of libraries available for I2C type displays. The one referenced here works with the particular LCD I used. If you use a different one, you might need 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 communications port. Note that the communications port can also be used for debugging using the *Serial Monitor* capability in the Arduino IDE, but not at the same time that the N1MM+ program is running. Also, note that the N1MM+ Rotor program is only capable of operating at 9600 baud.

I haven't tried it, but used the VSPE program from Eterlogic to allow 2 programs to simultaneously communicate with a radio using USB protocol; that might also work here.

- Compute the values for *timerDegrees* and *maxAzimuth* based on the definition of *PULSE_DEGREES* defined in the *My_Rotator.h* header file.
- Initialize the *currentAzimuth*, *targetAzimuth* and the flags that indicate the state of the rotator. The value of *currentAzimuth* is read from the Arduino's EEPROM, where it is periodically saved when the controller is in use.
- Configure the I/O pins used to control the rotator and make sure the two relays used to control the power and direction of rotation are both off.
- Initialize the liquid crystal display.
- Set up the encoder input pins.
- Define what to do when the encoder generates an interrupt.
- Define what to do when we get an index pulse interrupt.
- Set up the timer interrupt mechanism.
- Define the software reboot pin.

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:

- Sets the *systemTime* on each pass through the loop. We do this because the *millis()* function does not work correctly inside interrupt service routines.
- Calls the *Say_Hello()* function to announce that the program is running and looking for work to do.
- If the serial I/O port is connected to a computer, looks to see if the N1MM+ program (or serial monitor) has sent a command, and if so, reads it from the *Serial* stream.

- If the serial I/O port is connected to a computer, calls *Process_Command()*, to set up any variables or flags required to execute the command, or in some cases, it simply executes the command.
- Sets the *azMultiplier* variables to the appropriate value as determined by the position of the front panel toggle switch or lack of it. This is used in the *Read_Encoder()* function to compute the target azimuth change each time the encoder is moved.
- Calls *Display_Target()*, which displays the *targetAzimuth* on the first line of the display if it was changed by the command processing functions or via movement of the rotary encoder.
- Calls *Start_Rotator()*, which will initiate any movement required by the most recent command.
- Calls *Check_Timeout()*, to see if too much time has expired since receiving the last *MOTOR_INDEX* pulse from the rotator.
- Calls *Read_Pulse()* to see if a *MOTOR_INDEX* or timer generated pulse was received, and if so, update the *currentAzimuth* and reset the timeout.
- Calls *Check_Reversal()* to see if the N1MM+ program or the operator via manual control requested a new azimuth requiring the rotator to reverse direction.
- If the rotator is not currently turning, it calls *Update_Az()* to see if the EPROM should be updated.
- If the rotator is not turning, it calls the *Check_LCD()* function to see if anything other than the hello message needs to be displayed.
- If the operator operates the encoder pushbutton, we call *Calibrate()* to recalibrate the rotator.
- Calls *Ck_Reboot()* to see if the operator pushed the reset button on the back panel.

Note that *Check_Timeout()*, *Read_Pulse()* and *Check_Reversal()* are only called if the rotator *isRotating*. *Get_Command* and *Process_Command* are only called if a serial port is connected as that would be the only way to send a character command.

The Get_Command() Function

The *Get_Command()* function checks to see if any data has arrived at the *Serial* port, and if so, uses the *Serial.peek()* function to see what the first character is.

The *Serial.peek()* function doesn't remove the character from the buffer, but rather just sees what it is. It also returns a zero value if there is no data in the buffer.

If there are one or more characters in the buffer, the function reads the entire command into the *commandBuffer* for later use in the *Process_Command()* function. Note the use of the *Serial.readStringUntil* function to read the command line. The N1MM+ program uses a return character as a command line terminator instead of the more common newline character. If you're using a different controller program, that mighe need to be changed in this software.

Next the *Get_Command()* function searches the *commandTable* for the single character command and if found, returns the numerical value associated with the character.

If no match is found, the *commandBuffer* is cleared and a value of *NO_COMMAND* is returned.

Note, when using the serial monitor, the command letters may be entered in either upper or lower case.

The Process_Command() Function

Process_Command() takes care of setting up all the parameters needed to turn (or stop turning) the rotator based on the incoming command. In some cases (e.g., the *Stop* and *Azimuth* inquiry commands) this function actually executes the command.

For those commands that require the rotator to move, the general logic is that there are two azimuths maintained by the program. Any time the *targetAzimuth* and *currentAzimuth* are not equal, the *Start_Rotator()* function will turn the rotator. Thus the processing associated with rotator movement commands in here consists of simply setting an appropriate value of *targetAzimuth*. That could be the azimuth sent via the "M" command, or a full scale azimuth in the case of an open-ended rotate command or the calibrate command (neither of which are used by the N1MM+ program).

In the case of the *Stop* or *Cancel* commands, the *targetAzimuth* is set to the value of the *currentAzimuth* and the *isRotating* flag is set to *false* and the motor is stopped.

There is one command included that is not part of the GS-232 command set. I call it the *Rotator_Zap* command (‘Znnn’). It can only be sent from the *Serial Monitor*, not from the N1MM+ program. It is used to set specific equal values of *targetAzimuth* and *currentAzimuth* and to force an update of the *currentAzimuth* stored in the EEPROM. This is for debugging and testing purposes.

The Start_Rotator() Function

Start_Rotator() first checks to see if the rotator is already in motion and if the *currentAzimuth* and *targetAzimuth* values are equal. If either is true, then the function just returns.

If the rotator needs to be turned, we figure out in which direction. If the *targetAzimuth* is greater than the *currentAzimuth*, the rotator needs to turn counter-clockwise. If the *targetAzimuth* is less than the *currentAzimuth*, the rotator needs to turn clockwise.

Based on that, the correct setting for the direction control relay is set and the value of *deltaAzimuth* is set to either + or –*timerDegrees*.

The rotator motor is then turned on, the *isRotating* flag is set to *true*, the *pulseTime* and *debounceTime* are recorded and the new target and current azimuths are displayed.

The Check_Timeout() Function

If the rotator is not currently turning, the *Check_Timeout()* function doesn’t do anything.

If the rotator *isRotating*, we check to see if the difference between the current *systemTime* and the *pulseTime* is greater that the *ROTATOR_TIMEOUT* limit. If it is, we assume it is because the rotator hit the mechanical end stop.

If a timeout occurs, the motor relays are turned off and the *currentAzimuth* is set to either *maxAzimuth* or *minAzimuth* depending on which way the rotator was turning. *targetAzimuth* is then set to the value of *currentAzimuth*. The *isRotating* flag is set to *false* and a timeout message is sent to the display.

The rotator takes a little over 45 seconds to complete a 360° rotation, and based on observations, a *MOTOR_INDEX* pulse is received on the average about every 806mS. To allow a little extra time, the *ROTATOR_TIMEOUT* is set to 850mS.

If a timeout occurs, we also flash the azimuth saved LED for about 1 second and save the now calibrated azimuth.

If a timeout occurs, the function returns a value of *true*. Otherwise it returns *false*. The return code is only used in the *Calibrate()* function.

In the process of testing Version 3.0, I discovered that there are actually more than 360° between the CW and CCW end stops in the rotator. To compensate, I added the *Backup()* function, which we use here to move the rotator away from the end stop approximately *CAL_ADJUSTMENT* degrees whenever it hits the stop

The Timer_ISR(), Pulse_ISR() and Read_Pulse() Functions

Logically, these three functions are part of the same processing logic associated with dealing with incoming *MOTOR_INDEX* pulses and timer generated pseudo pulses.

Timer_ISR()

Timer_ISR() is invoked by the Arduino’s interrupt handling mechanism by the *TimerOne* library logic every *timerInterval* mS (140mS for my rotator).

How is the 140mS number determined? This is important, because it might need to be a little different for different rotators. My rotator takes just a little over 45 seconds to complete a full 360° rotation. Based on timing tests for that particular rotator, index pulses from the cam switch are received on the average about every 806mS. That means it takes about 134mS for my rotator to turn 1°.

The value set for *timerInterval* needs to be slightly more than the time required for 1° of rotation (in my controller, it's 4% more). This makes certain that the controller will see no more than 5 pseudo pulses from the timer before seeing the hard index pulse from the cam switch. If the value of *timerInterval* is too low, the controller might see 6 timer pulses between cam switch pulses, and thus, it will get totally out of calibration quickly.

Conversely, if the value of *timerInterval* is too high, the controller might only see 4 timer pulses between real index pulses, and again, will get out of calibration quickly. The closer the *timerInterval* value is to 1/6 of the hard pulse time, the more accurate the intermediate readings will be.

The Version 1.0 software, which did not use the timer interrupts will run on later versions of the hardware. It contains conditionalized code that will allow you to determine the minimum, maximum and average cam pulse time intervals for a particular rotator.

The function is pretty simple. If the rotator isn't actually turning, it ignores the interrupt. If the value of *pulseType* is set to *HARD_PULSE*, it ignores the interrupt. Why this test, one might ask? Since the value of *timerInterval* is set to generate interrupts at slightly more than 1° intervals, we expect to get the 6th timer interrupt in a sequence slightly after seeing the real index pulse. This test indicates that a real pulse is waiting to be processed, and thus we should ignore the pseudo pulse.

If the above tests are both false, the value of *pulseType* is set to *SOFT_PULSE* and we're done.

Pulse_ISR()

The whole approach to processing the received hard index pulse may seem to be a bit convoluted because I managed to handle some issues in the software as opposed to adding additional hardware to the interconnect board.

The cam switch in the rotator has a lot of contact bounce, and thus doesn’t send a single pulse, but a series of them everytime the switch is activated. There are some debounce libraries available on GitHub and other places on the internet, but all the ones I looked at seemed to be designed to deal with switch bounce on buttons operated by humans, where after the initial transition, there will be a series of false transitions followed by the switch remaining in a new stable state for some relatively long period of time (at least in terms of processor instruction cycles).

The hardware modifications in Version 5.2 have pretty much eliminated the contact bounce problem.

The cam switch in the rotator sends an AC pulse which is anywhere from about 40 mS to 100 mS in length depending on the mechanical adjustment of the switch in the rotator. In Version 5.2, the AC pulse is rectified and filtered by a 10uF capacitor. The filtered pulse is then run through a solid state relay the output of which drives the interrupt enabled digital pin 18 of the Arduino.

In the *setup()* function we set up the interrupt to trigger on the trailing (rising) edge of the pulse. The logic here is to react to the initial state change and then ignore any other state transitions for a period of *PULSE_DEBOUNCE* mS (currently set to 15mS), although the new hardware configuration has almost completely eliminated any contact bounce problems as shown in the scope trace earlier in this document.

So what we do in the *Pulse_ISR()* function is quite simple. We check to see if the variable *debounceTime* is non-zero. If it is, that indicates that we have already seen the first state change in the series of pulses and the function ignores the interrupt.

If *debounceTime* is zero, this is the first state change, so we set it to *systemTime* and also set *pulseTime* (used in the timeout logic) to *systemTime*. We use *systemTime* (updated on every pass thorough the *loop()* function) instead of a call to the *millis()* function as that function doesn’t work correctly inside interrupt service routines.

We also set the value of *pulseType* to *HARD_PULSE* and reset the interrupt timer to zero.

Read_Pulse()

The *Read_Pulse()* function, which is invoked from the main loop, first checks to see if the rotator *isRotating*. If not, it simply returns. As a double check, if the value of *pulseType* is *NO_PULSE*, it also returns.

If the value of *pulseType* is *HARD_PULSE*, we see if the *PULSE_DEBOUNCE* time limit has expired. If not, the function again, just returns. When this happens, we aren't ignoring the pulse but rather deferring processing it until the *PULSE_DEBOUNCE* time expires. That works because we don't zero out the *debounceTime* until the pulse has been processed, which is what keeps the *Pulse_ISR()* routine from resetting the pulse times.

Once the *pulseTime* expires or if the *pulseType* was *SOFT_PULSE*, we process the pulse, which is pretty simple; we increment or decrement the *currentAzimuth* based on the value of *deltaAzimuth* set in the *Start_Rotator()* function, zero out the *debounceTime* and set the *pulseType* to *NO_PULSE*.

Since we saw a good pulse, the value of *lastGoodAzimuth* is set to the *currentAzimuth*.

If the *Tgt_Reached()* function returns *true*, the rotator is stopped and the *isRotating* flag is set to *false* and the new *currentAzimuth* is sent to the LCD, and the *targetAzimuth* is set equal to the *currentAzimuth*.

The Tgt_Reached() Function

The *Tgt_Reached()* function first compares the *currentAzimuth* to the *targetAzimuth* and returns *true* if they are equal.

Because of the goofy azimuth math in Version 3.0, sometimes the *currentAzimuth* was exceeding the minimum and maximum limits, so the function checks to see if that is the case, and if so, sets the *currentAzimuth* to either *minAzimuth* or *maxAzimuth* appropriately.

However, if the rotator *isCalibrating*, we don't do the limit test, as when it is being calibrated, the software forces it to rotate beyond the limits.

The Check_Reversal() Function

The *Check_Reversal()* function handles the issue of the operator changing the *targetAzimuth* to a value that would require the rotator to move in the opposite direction from the direction in which it is currently moving. This can be done either via a command from the N1MM+ program or via physical movement of the rotary encoder.

The logic is simple; if the rotator isn't moving, we do nothing. If it is rotating, we look to see if the current direction of rotation needs to be reversed based on a comparison of the *targetAzimuth* and *currentAzimuth*. If the direction of rotation does need to change, we simply stop the rotator and set the *isRotating* flag to *false*. Because the *targetAzimuth* and *currentAzimuth* are still not the same, the next call to *Start_Rotator()* will start it moving again in the proper direction.

Note that this approach increases the probability that the rotator will come to a complete stop before starting it in the opposite direction, thus reducing the mechanical stress on the rotator itself.

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 hardware interrupt capability.

The first thing the *Read_Encoder()* function does is to look at the state of the *isCalibrating* flag. If the rotator is being calibrated, we don't want the operator to be able to change the *targetAzimuth* while that is happening. Since the *Calibrate()* function takes control away from the main loop, the operator won't be able to change the *targetAzimuth* from the N1MM+ program or the serial monitor either.

The encoder library function *process()* takes care of determining which direction the operator is turning the knob. Depending on which way the knob is turning, we either increment or decrement the *targetAzimuth*.

In Version 4.0, the amount that the *targetAzimuth* is changed depends upon the setting of the front panel toggle switch (if installed).

Note that we also never let the *targetAzimuth* exceed the minimum or maximum limits.

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.

Another thing to note is that on some of the encoders available on the internet, the 'A' & 'B' pins are reversed, which makes the *Rotary.process()* function think that the encoder is moving in the opposite direction from what it is really doing. The problem is most easily fixed by simply reversing the pin assignments (in the *My_Rotator.h* file) in the software. The encoders I happened to have on hand have this issue, and thus you will note the pin designations are reversed in my software.

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. the *targetAzimuth* reading or an 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 *helloTimeout* 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 indicating that the hello message is, in fact, being displayed.

The Display_Rotation() Function

The *Display_Rotation()* function lets the operator know the rotator is turning by updating the *currentAzimuth* on the second line of the display. Note that it does not re-write the entire second line, but rather, just changes the digits.

Note that in Version 3.0, because the azimuths are not nice multiples of 1° that the number stored internally is rounded off before being converted back to a real azimuth to be displayed.

The Display_Target() Function

The *Display_Target()* function displays the *targetAzimuth* on the first line of the display whenever the rotator is turning (except during calibration).

Again as in the case of the *Display_Rotation()* function, the internal azimuth is rounded off before 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.

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 would remain on the display.

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

The Int_2_Az() Function

The *Int_2_Az()* Function is used whenever we need to display an azimuth reading on the LCD or respond to the “C” command from the N1MM+ program.

This function builds a *String* containing the ASCII representation of the real azimuth. The *String* is always a 3 digit number padded with leading zeros if necessary.

Note that the Azimuth argument is expected to be a real azimuth in the range of 0° to 360°, not the internally maintained azimuth.

The Update_Az() Function

The *Update_Az()* checks to see if the value of the *currentAzimuth* needs to be updated in the EEPROM. We do not do this every time the *currentAzimuth* changes for two reasons. First, it takes a really long time to write to the EEPROM. Secondly, the EEPROM has a limited number of times it can be written to before it fails. That is a huge number, but if we were to write the *currentAzimuth* every time it changed, we could possibly hit that limit over time.

Thus, we only update it after there has been no rotation for *EEPROM_TIMEOUT* milliseconds.

An important note; after the controller has been in use for a while, before turning it off, the operator should make sure that the *currentAzimuth* has been saved as indicated by the green LED on the front panel. If the unit is turned off without the azimuth having been saved, it will be considerable out of calibration the next time it is turned on.

If the controller in use has the Version 2.1 software and associated wiring changes, pushing the “Reset” button will force the azimuth to be saved.

The Calibrate() Function

The *Calibrate()* function can be invoked from the command processing logic, however, the N1MM+ program does not use the command for a Yaesu rotator. The function is called, however from the main loop if the pushbutton switch built into the rotary encoder is operated.

Also note that the rotator is pretty much self-calibrating. Anytime a timeout occurs, it is assumed to be due to the fact that the rotator hit the mechanical end stop. Anytime that happens, the *Check_Timeout()* function sets the azimuths appropriately.

Note that the function takes over total of control from the normal processing in the main loop, so any commands sent from the N1MM+ program will be ignored until the calibration process has completed.

After setting the *isCalibrating* flag to true (which causes *Read_Encoder()* to ignore any interrupts), we see which end stop we think we are closer to and set the *targetAzimuth* to either *minAzimuth* minus *maxAzimuth* or *maxAzimuth* times 2 so as to minimize the time to perform the calibration and ensure that the rotator actually hit the end stop. Note that the assumed direction of rotation needed may or may not be correct depending on how far out of calibration the rotator is.

In previous versions the value of *currentAzimuth* was used to make this determination, however if a timeout occurred due to just missing a cam pulse that was not a result of hitting the end stop, the value in *currentAzimuth* could be way off. To fix this, in Version 4.1, a new variable was introduced (*lastGoodAzimuth*) which is only set after seeing a good pulse. This new variable is used to determine which way to turn to do the calibration.

The function mimics the rotator movement logic of the main loop, except we are looking for and expecting a timeout to occur.

When the timeout occurs, we assume that the rotator hit the mechanical end stop, and the logic in the *Check_Timeout()* function will set the correct values of *currentAzimuth* and *targetAzimuth* and stop the motor.

If the timeout occurred, we clear the *isCalibrating* flag, inform the operator that the rotator is calibrated, and we’re finished.

The Start_Motor() Function

Start_Motor() is pretty simple. We set the direction of rotation based on the value of the *Dir* argument then after a 10mS pause to allow the contacts in the direction relay (‘K2’) to settle, we activate the power relay (‘K1’).

Once the motor is running, the function sets the appropriate values for the index pulse handling and timeout parameters.

The Stop_Motor() Function

Stop_Motor() turns the power relay off and after a 10mS pause to allow ‘K1’ to settle down then turns the direction relay (‘K2’) off.

Once the motor is stopped, the function clears the values for the index pulse handling and timeout parameters.

Note, the function accepts a *String* type argument which will be displayed in the serial monitor if debugging is enabled.

The Backup() Function

The *Backup()* Function was added in Version 2.2. In testing, I discovered that there are more than 360° between the CW and CCW end stops on the rotator. This function is called from the *Check_Timeout()* function when the rotator does hit the stop, and using a *Time* parameter (in milliseconds), backs the rotator off the stop by approximately *CAL_ADJUSTMENT* degrees. It does this on the QT; that is, it doesn’t mess with the azimuth readings or any of the flags associated with normal rotation.

I was also thinking that maybe this should be used whenever the rotator stops due to a *HARD_PULSE* being received to back it up by the debounce masking time, but so far, haven’t done so.

The Ck_Reboot() Function

Ck_Reboot() looks to see if the reset button on the back panel is operated, and if so, saves the *currentAzimuth* in the EEPROM and performs a software initiated reboot.

The Blink_LED() Function

Whenever a timeout occurs, either because the rotator hit the end stop unexpectedly or during calibration or when the operator pushes the reset button, this function flashes the azimuth saved LED for a second.

Tuning the Timing Parameters

Throughout the description of how the software works, there are many places where I mention the values of the various parameters associated with the timing and rotation of the rotator. I figured it would be useful to the reader to consolidate how I arrived at the various numbers in one place.

In order to achieve the most accuracy possible, it is important to tweak these parameters for a particular rotator; again, this is really only necessary if you're using it on a very narrow pattern antenna; it's not really going to matter for something like a tribander.

There are three parameters that you can adjust are:

- *timerInterval* – Time in mS between soft (timer) pulses
- *PULSE_DEBOUNCE* – delay time after cam (hard) pulse is detected
- *ROTATOR_TIMEOUT* – Time when we assume we missed a hard pulse

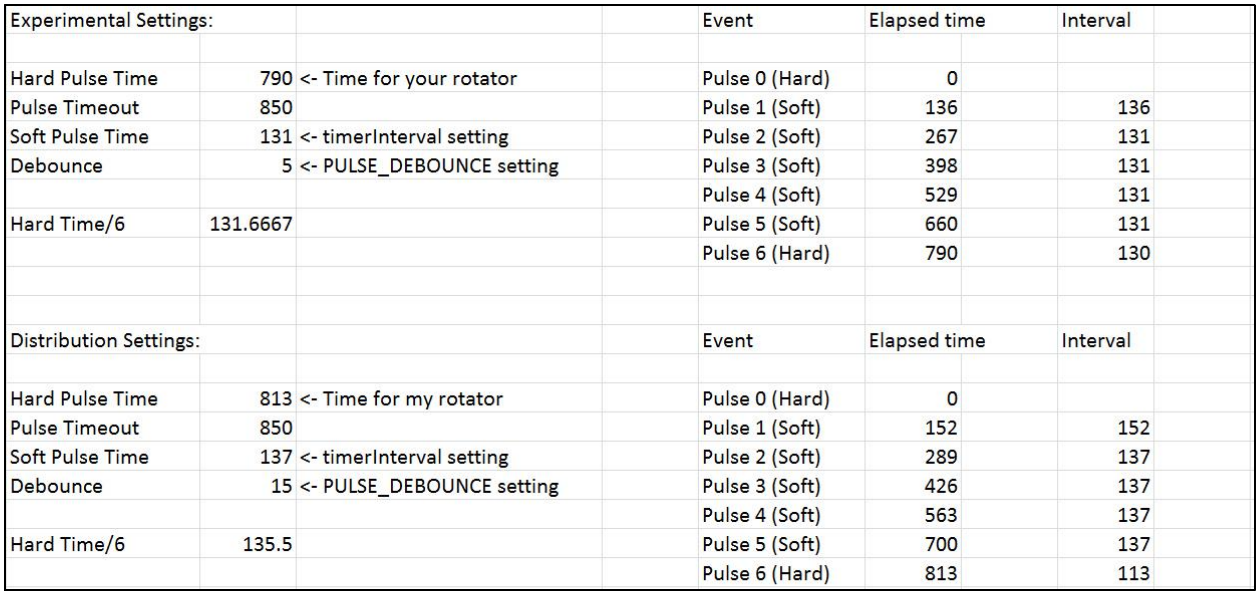
Included in the distribution package is an Excel spreadsheet (Timing.xlsx) which you can use to really fine tune these parameters for your particular rotator.

But before you start, understand a few assumptions:

- Because the amount of rotation between cam pulses is all based on mechanical stuff, we assume that is fairly consistent for all AR-22 rotators.
- The computations in the code are all based on the amount of time between cam switch pulses, which is based on rotational speed. It might be that the rotational speed could vary slightly based on the temperature; I haven't tested this theory.

  Also, different rotators seem to run at slightly different speeds; one of mine turns about 3% faster than the other one.

Here’s what the spreadsheet looks like; all the numbers are milliseconds:



Experimental Settings:			Event	Elapsed time	Interval
Hard Pulse Time	790	<- Time for your rotator	Pulse 0 (Hard)	0	
Pulse Timeout	850		Pulse 1 (Soft)	136	136
Soft Pulse Time	131	<- timerInterval setting	Pulse 2 (Soft)	267	131
Debounce	5	<- PULSE_DEBOUNCE setting	Pulse 3 (Soft)	398	131
			Pulse 4 (Soft)	529	131
Hard Time/6	131.6667		Pulse 5 (Soft)	660	131
			Pulse 6 (Hard)	790	130
Distribution Settings:			Event	Elapsed time	Interval
Hard Pulse Time	813	<- Time for my rotator	Pulse 0 (Hard)	0	
Pulse Timeout	850		Pulse 1 (Soft)	152	152
Soft Pulse Time	137	<- timerInterval setting	Pulse 2 (Soft)	289	137
Debounce	15	<- PULSE_DEBOUNCE setting	Pulse 3 (Soft)	426	137
			Pulse 4 (Soft)	563	137
Hard Time/6	135.5		Pulse 5 (Soft)	700	137
			Pulse 6 (Hard)	813	113

The parameter values in the bottom section are the settings in the software as distributed, and they work for both of my rotators.

The data in the top section shows the settings that optimized the performance of the test rotator in my basement. Notice that that one turns a little faster than the one on my roof (790 mS between cam pulses as opposed to 813 mS).

If you look at the rightmost column, you will see that the intervals between pulses with the settings in the distributed software are not all even, whereas those for the test rotator are. Those settings did not work for the slightly slower rotator on the roof.

So to tweak the numbers for your particular rotator, first you need to measure the interval between cam pulses (and note the software triggers off the end of the pulse, not the beginning; there’s a reason for that). The scope trace earlier in the document shows that the hard pulse interval for my test rotator is ~790 mS. Dividing that by 6 gives the number of milliseconds for one degree (actually 0.974°; see next section) of rotation as 131.667 mS, so for that rotator, I set the *timerInterval* variable to 131000 and it worked great.

The timing between the initial hard pulse and the first soft pulse is *PULSE_DEBOUNCE* milliseconds longer than the rest as the soft pulse timer is not reset until the pulse is actually processed.

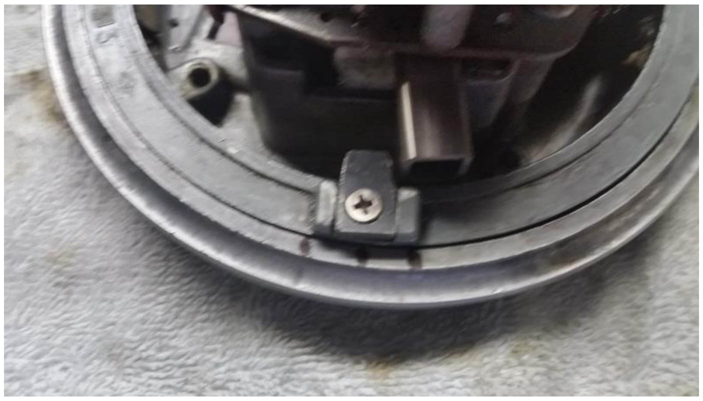
Once you have everything else working, you can experiment with reducing the *ROTATOR_TIMEOUT* setting although that's not really critical.

And just for the record, if I didn't already mention it somewhere else in the document, the reason the *Pulse_ISR* interrupt is triggered from the trailing edge of the pulse versus the leading edge is the solution to the problem of the rotator hitting the mechanical end stop with the cam switch operated.

Rotational (Cam Switch) Error

Running the rotator using the original clunk-clunk box, I discovered that after commanding the rotator to turn 360°, it was coming up consistently short of turning 360°.

To determine the amount of the error, with the motor assembly removed from the rotator, I marked the rotator as shown in the following picture:



The two outer marks represent the inner sides of the stop arm, and the center mark is halfway in between, which I consider to be the 0°/360° point. Note, the stop pin in my rotator is about twice as wide as the stock one, as mine broke off and I repaired it as shown in the picture. I also had to build a new stop arm, as the original one got mangled. Notice the new one is made of a piece of ½" square tubing instead of the ½" x 3/8" U shaped piece that was originally used. I highly recommend this modification whether needed or not. I have 2 AR-22 rotators and the stop arms in both were bent out of shape.

Replacing the ring gear with one made of stainless steel is also a good idea. In my #2 rotator, the ring gear was cracked and a shown above, on the #1 rotator, the stop pin broke off.

Using the center mark as a reference, after turning the rotator 360° according to the clunk-clunk box, I measured the distance between where the pin stopped and the center mark. For my rotator, the average error was 0.47”. The diameter of the ring gear is 5.75”, and thus its circumference is 18.06”. In degrees, the amount of error is:

    360 x 0.47 / 18.06 = 9.39°

That means that for 60 clicks on the clunk-clunk box, the rotator is only turning 350.61°. The number of cam switch pulses for a full 360° rotation is then:

    60 x 360 / 350.61 = 61.61 Pulses

Multiplying that by 6 gives the total number of pulses (cam plus timer) need for 360° of rotation as 369.64.

Dividing that by 60 gives us 5.844° for each cam pulse, and dividing that by 6 gives 0.974° for timer pulses.

In the *My_Rotator.h* file for my rotator, I set the *PULSE_DEGREES* definition to 5.85 degrees. The *timerDegrees* value is computed based on that in the *setup()* function.

Timeout Backup Adjustment

The final variable that will need to be tweaked is the value of the *CAL_ADJUSTMENT* definition in the *My_Rotator.h* header file. This value defines the approximate number of degrees to back up the rotator whenever it hits the end stop (either in normal operation or during calibration).

In my version of the code, it is set to 5°. But, as shown in the picture above, my stop pin is about twice as wide as the original, which means that in most cases, the value might have to be increased, although I didn’t bother doing that for the rotator with the standard stop pin and it didn’t really seem to matter.

Operation

Calibration

Before using the controller, if your rotator is not yet mounted somewhere, you should set it up in the workshop and do the tests described above in the “*Tuning the Timing Parameters*” section. I’ve built 2 of these controllers, and I’ve tested them with two rotators, one of which, as mentioned has a wider stop tab than the stock rotator. I can’t guarantee that the values I have set for all the timing parameters will work correctly for all rotators.

If you can’t set it up in the workshop and there seems to be issues with the timing parameters, you can still play with them in the software to see if it makes any improvement.

Using It

The operation should be pretty much self-explanatory, but I’ll give you a few of the highlights here.

The first time you turn it on, it may read some random number from the EEPROM address where the current azimuth is saved, and thus the first thing to do is to calibrate the controller with the rotator by pressing the encoder switch. The unit will announce that it is “Calibrating” and when finished, it will announce that it is “Calibrated”. Now you’re ready to go.

You can use the (Version 4.0) azimuth increment toggle switch to set the number of degrees that the target azimuth should be incremented or decremented for each click of the encoder. I keep mine set on 5° as I’m using it with a 3 element 6 meter beam, so the azimuth need not be that accurate. If you’re turning a highly directional UHF array, then maybe you would want to use the 1° setting. Note that in the 1° mode, it takes a lot of turns of the encoder to move the antenna any significant amount.

When you move the encoder, the new target azimuth will be shown on the top line of the display, and you will see the current azimuth changing on the 2nd line. When they are equal, the rotation will stop.

10 seconds after the rotation stops (you can change this in the header file) the display will show “Azimuth Saved” on the top line for 3 seconds and the green “Azimuth Saved” LED will light (if you installed it).

Should the rotator prematurely hit the end stop, the controller will stop seeing pulses from the cam switch and will show “Timeout” on the top line of the display. If the “Azimuth Saved” LED is installed it will also flash for about 1 second. When a timeout occurs, the current azimuth will be set to either 0° or 360° based on which way the rotator was turning when the timeout occurred.

Known Bugs and Glitches

If the USB connection drops, when the AR-22 controller is being used with the N1MM+ program (which I’ve only seen happen when I accidently knocked the connector out of the laptop), the N1MM+ program won’t re-establish the connection with the AR-22 controller when the USB cable is plugged back in.

The only cure seems to be to exit the N1MM+ program, reset the AR-22 program using the Reset button on the back panel (which will save the *currentAzimuth* in the EEPROM) then restart the N1MM+ program.

Note, if the controller is wired according to the Version 1.0 or 2.0 schematics, the Reset button does NOT save the azimuth, so make sure it has been saved before performing the reset.

This problem has been reported on the N1MM+ Yahoo group, but no action seems to have been taken to solve the problem.

It’s not a bug, it’s a feature! But the operator might think it’s a glitch; when the rotator is being calibrated, it is possible for the display to show a current azimuth of more than 360°. It could also show an azimuth of less than zero, however, the software will convert negative readings to positive azimuths of slightly less than 360°. For example, if the internally maintained *currentAzimuth* is -006°. It would be displayed as 354°. This is the only case where the azimuth displayed can be outside the 0° to 360° range. It is possible when calibrating, because the rotator itself is not in agreement with the azimuth in the software.

Coming Attractions

Hardware Enhancements

One hardware enhancement I'm thinking about, although it looks like a lot of work and at least two more wires in the cable, is to replace the mechanical end stop in the rotator with limit switches (like the heavier duty rotators have). The stop mechanism in my test rotator actually broke although I don't think it was due to the behavior of the smart controller, but rather simply old age. The modification would require some slight software changes.

Support for Other Rotators

As previously mentioned, this controller hardware and software would probably work with some slight modifications for the Alliance rotators or any others that use a clunk-clunk type controller.

I looked at the CDE Ham-M rotator manual to see if this controller could be adopted to run that rotator. The simple answer is "No"! It would require totally different hardware and a lot of changes to the code.

Having said that, it would be possible to build a similar unit for the Ham-M, and that design would probably also work with the Hy-Gain Ham-IV rotator, which has almost identical circuitry to the Ham-M.

Bill of Materials

Parts needed for the main controller circuit board (optional IC sockets are not included):

Processor	Arduino Mega 2560
C1, C2	10uF – 50V electrolytic capacitor
C2	0.22uF – 50V capacitor
D1	1N4002
R1	10Ω – 1/4W

R2	470Ω – 1/4W
R3	220Ω – 1/4W
J1	4 pin female vertical header
J2	3 connection screw terminal – 0.2” pin spacing
J3	2 pin right angle male header
J4 – J7, J9	Various pin count right angle male headers; the PCB is designed so that a single 16 pin header can be used for all.
J8	4 connection screw terminal – 0.2” pin spacing
J10	4 pin vertical male header
3 ea.	8 pin female vertical header with long pins; connections between the controller board and the Arduino.
1 ea.	10 pin female vertical header with long pins; connections between the controller board and the Arduino.
K1, K2	Fujitsu NA-5W-K 5V DPDT relay.
U1	LCA-110 solid state relay

These parts are needed for the power supply circuit board:

C1	470uF – 50V electrolytic capacitor
C2, C3	0.01uF (100nF) ceramic disk capacitor
D1, D2	1N4002
R1	1.8KΩ ¼ watt resistor
J1	2 pin male vertical header
J2	4 pin male vertical header

J3	8 pin female vertical header with long pins; You actually need 2 of these as shown on the PCB layout. The 2nd one is installed next to the voltage regulator. It has no electrical connections, but is for mechanical stability.
U1	LM7812 voltage regulator (TO-220 package) with additional heat sink

The following parts are not shown on the above schematics, but rather mount on the enclosure used. Many of them connect to the various headers on the controller board or power supply board. The pictures should help you figure it out:

F1	2 amp fuse and appropriate holder
S1	Power switch
S2	Azimuth switch; SPDT – center off toggle switch
S2	Run/Update switch; SPST toggle switch
S3	Reset button; SP pushbutton
T1	28V – 2A – CT; Hammond model 187E28 or equivalent
C1	Motor starting capacitor - ~145uF – 165VAC
Encoder	Keyes KY-040 Rotary Encoder
Display	I2C 1602 2 line by 16 character LCD display
D1	General purpose 20mA red LED (power)
D2	General purpose 20mA green LED (azimuth saved)
Rotator Connection	My controller has both a 4 conductor Cinch-Jones type connector and a barrier terminal strip installed for the connection to the rotator cable. Do whatever suits your personal needs.
Power Connector	I used the type used for PCs as I have a ton of PC power cables on hand!

USB Connector	This is a panel mount female type “B” to male type “B” USB cable. If you want to be able to load software without opening the enclosure or use the controller with PC based software (such as N1MM+), you’ll need this.