A Smart Controller for SPID Rotators
Version 1.2

©2024 John Price – WA2FZW

Table of Contents

Introduction

After several attempts to build an elevation/azimuth capable rotator system out of a pair of ancient Alliance C-225 rotators, I finally gave up and purchased an SPID-RAS elevation/azimuth rotator and when my old AR-22 failed, I also picked up an SPID-RAK azimuth only rotator.

The motors in these rotators run on 12VDC (but can run on up to 18 volts) unlike the motors in many other rotators which run on AC. This makes it much easier to precisely control both the elevation and azimuth angles.

The position is reported by the closure of reed switches built into both motor units that close once per degree of rotation or elevation. There are versions of the SPID rotators that have 0.5° or even 0.1° accuracy; the software only handles the 1° versions.

Rather than spend a lot of dollars on the stock controller, I decided that a modified version of the controller I built several years ago for the old CDW AR-22 rotator could be used instead.

There are four major components; a power supply, the controller PCB, the processor and a panel containing the switches used to set the azimuth and elevation.

This document describes the hardware and software for the new design.

Modification History

The original version of the software was designed specifically for the SPID-RAS Elevation/Azimuth rotator.

Version 1.1

In Version 1.1, I added the symbol *ELEVATION*, which when set to *false* disables all of the elevation functions allowing the code to be used with an azimuth only rotator such as the SPID-RAU or SPID-RAK.

There were no hardware changes to the controller or power supply circuits.

I reduced the height of the switch PCB and front panel PCB slightly from what is shown in the final build here and I added an alternate switch PCB and front panel PCB for the azimuth only implementation.

Version 1.2

In Version 1.2, the calibration procedure was completely overhauled. The new procedure is explained in the *Calibration* section of the document.

In previous versions, interrupts were used to detect index pulses from the motors. That approach simply wasn't working and was causing slight errors in positioning which would accumulate over time. In Version 1.2 I eliminated the use of interrupts. The *CheckPulse* function now handles all of the pulse processing.

The Power Supply

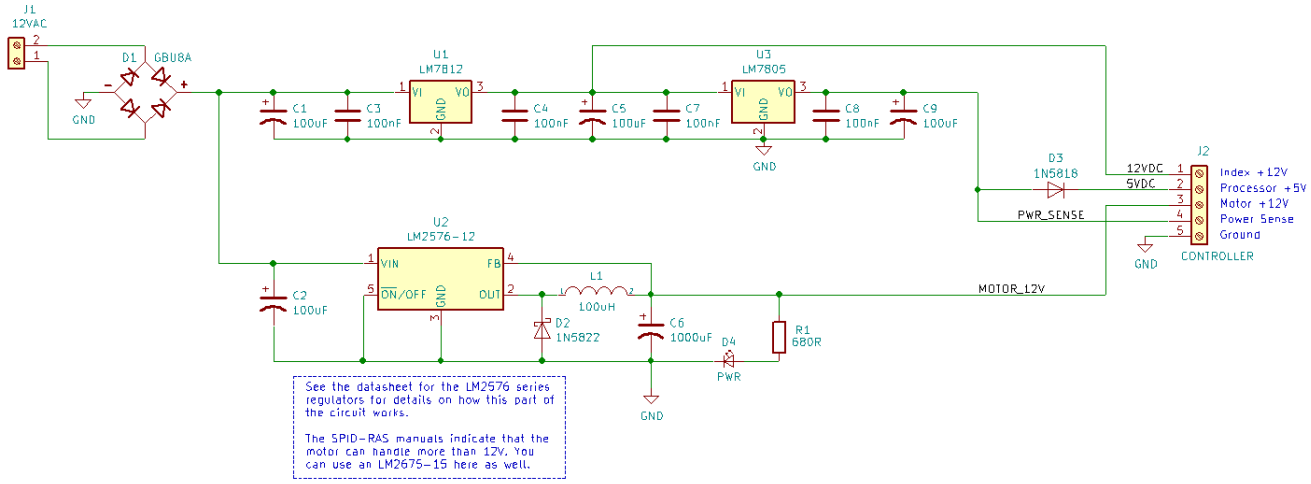
I designed the power supply to be pretty generic, in that it could be used for other projects.

The power supply supplies several voltages to the controller proper:

- 5VDC to run the processor, display and two of the motor control relays
- 12VDC (or 15VDC) at 3 amps to power the motors
- A separate 12VDC source to feed the pulse switches in the motor units and the *OPERATE* relay.
- A 5V output which provides an indication to the controller that the power supply is actually turned on (*PWR_SENSE*).

  This is needed when the Arduino is also connected to a computer via the USB connection. In that case, the processor is still powered via the USB connection and therefore still running. In that state, it is still capable of processing certain commands, but the software will prevent it from executing any commands that would cause the motors to run. For example, even with the motors disabled, the processor can respond to a request for the current azimuth and elevation.

Here’s the power supply schematic:



Not shown are external components such as the power switch, main fuse and power transformer.

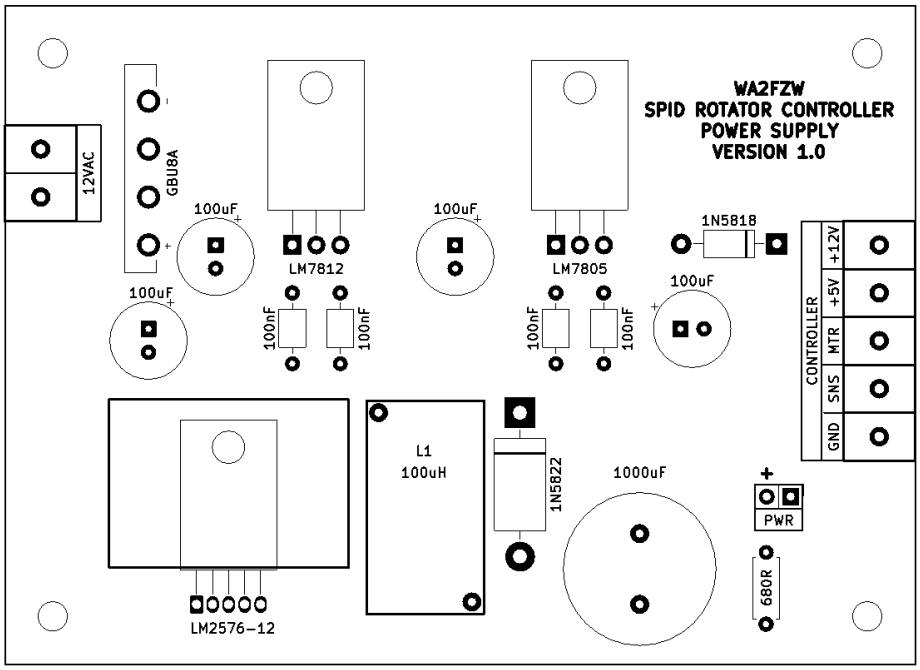
The GBU8A bridge rectifier is capable of providing 8 amps; the motors in the rotator only require about 2 amps and the current requirements for the processor and pulse switches in the rotator are only milliamps.

Notice there are two distinct regulator chains emanating from the rectifier. The LM7812 and LM7805 provide power to the pulse switches in the rotator and to the processor and display.

The second 12V supply uses an LM2576-12 regulator (or optionally an LM-2576-15). These are switching type regulators capable of providing 3 amps of current. The circuit used here was taken directly from the datasheet.

D5 is a Schottky diode. When the controller’s processor is connected to a PC, it would send 5V back into the power supply without the diode. The *PWR_SENSE* signal is taken before the diode so that the 5V coming from the processor when connected to the PC won’t be seen on the controller’s *PWR_SENSE* GPIO pin.

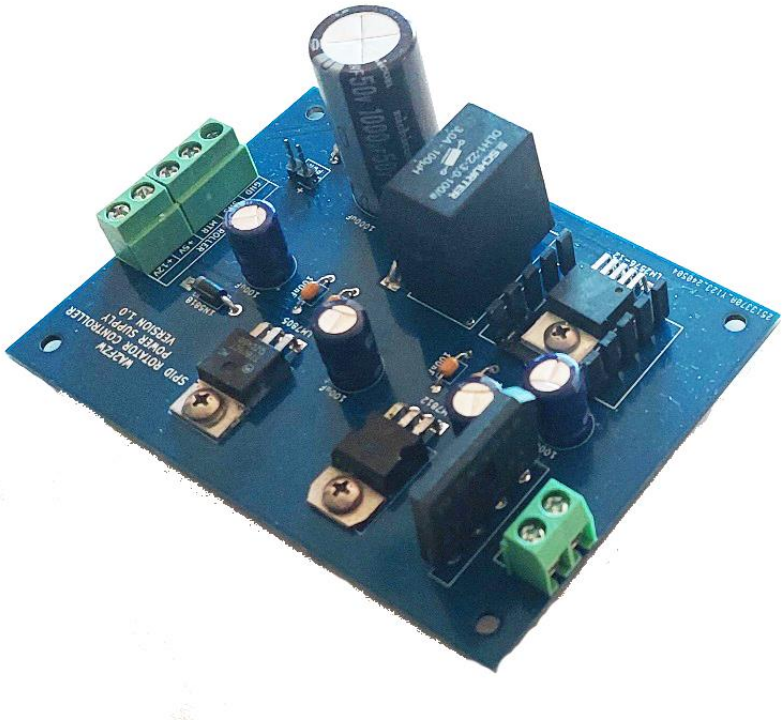
Here’s the component layout for the power supply PCB:



I left room around the LM7812 and LM7805 for heat sinks; however they are not really needed. I did use a heat sink for the LM2576-12 or LM2576-15 however it doesn’t seem to even get warm.

The 2 pin vertical header labeled *PWR* connects to the power LED assumed to be mounted on the front panel.

Here’s the assembled power supply PCB:



Motor Voltage Options

The SPID-RAS manuals indicate that the motors can handle more than 12V. Also when long cable runs are used, one might want a higher voltage to compensate for losses in the cables. The LM2576-12 can be replaced with an LM2576-15 if more voltage is desired. I am using the 15V regulator in my builds.

Also note that whereas the schematic shows a 12V transformer, I’m using a 14V one (with either regulator option).

As might be expected, there are operational differences depending on the voltage as summarized here. Note that these measurements were made with about 10 feet of cable between the rotator and the controller and no weight on the rotator. When I installed the SPID-RAK on the roof with the antennas I did not notice any appreciable difference in the 15V times.

	12V Motor Power	15V Motor Power
Rotation Speed	2 ½ minutes for 360°	2 minutes for 360°
Index Pulse Rate	410 milliseconds	340 milliseconds
Index Pulse Width	70 milliseconds	52 milliseconds
Actual voltage when motors are running (at the controller)	10.5V	13.6V

The variations caused by the different rotation speeds require some of the symbol definitions in the software to have different values. The differences will be discussed in the *Software* section.

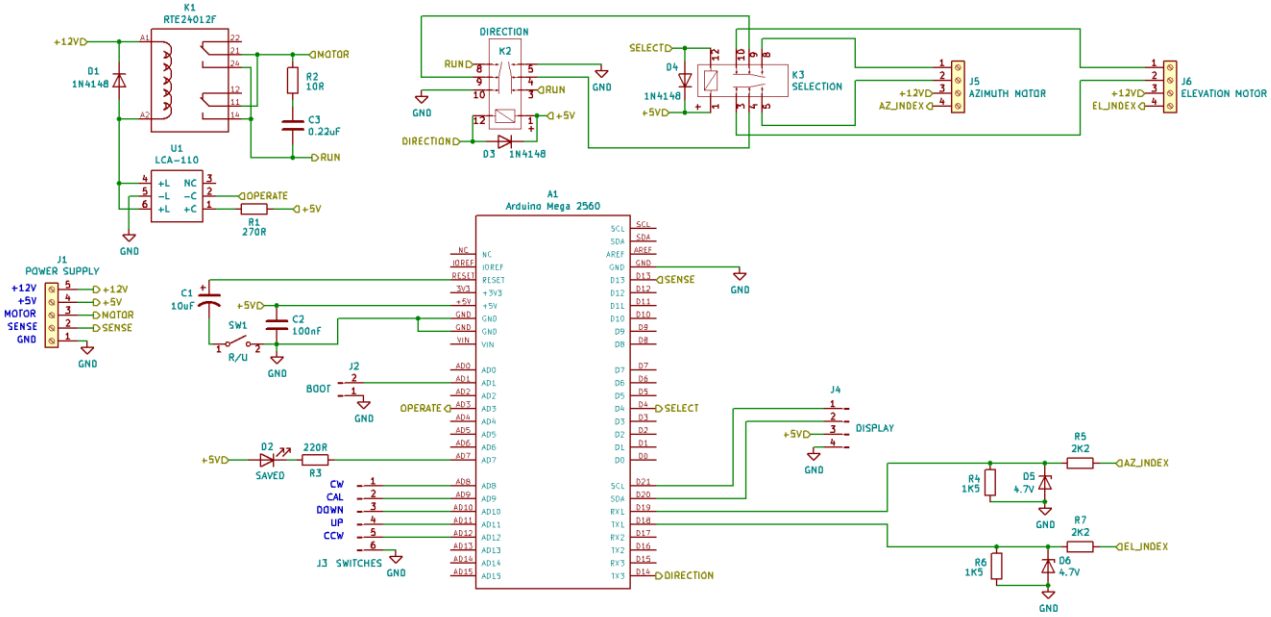
The Controller

There are several major components of the controller:

- The Arduino Mega2560 R3 processor.
- PCB containing the actual controller circuitry.
- A PCB containing pushbutton switches to control the azimuth and elevation.
- A 4 line by 20 character wide liquid crystal display.

The Controller PCB

Here is the schematic for the controller PCB:



The motors are controlled by the states of relays K1, K2 and K3.

K3 selects which motor to operate; the software only operates one at a time as the power supply is rated at 3 amps and the motors are rated at 2 amps. Only running one at a time also simplifies the software.

K2 selects the polarity to be applied to the motor to select the direction of movement and K3 selects which motor to run.

Only after the software selects the correct settings for K2 and K3, K1 is operated after a slight delay to actually send power to the selected motor. The combination of R2 and C3 is a ‘snubber’ to absorb any voltage kickback when the power is removed from the motor to prevent arcing of the relay contacts.

Two other sub-circuits on the board handle reading the switch pulses from the motors. The rotation speed of both motors is approximately 2.4 degrees per second, so the pulses come about every 410 milliseconds[1]. The power to the pulse switches in the rotator is +12V. There are two voltage dividers and Zener diodes (R4, R5, D5 and R6, R7, D6) that limit the pulse voltage seen by the processor to 4.7V. The outputs of the voltage dividers are read by two GPIO pins on the processor (D18 & D19).

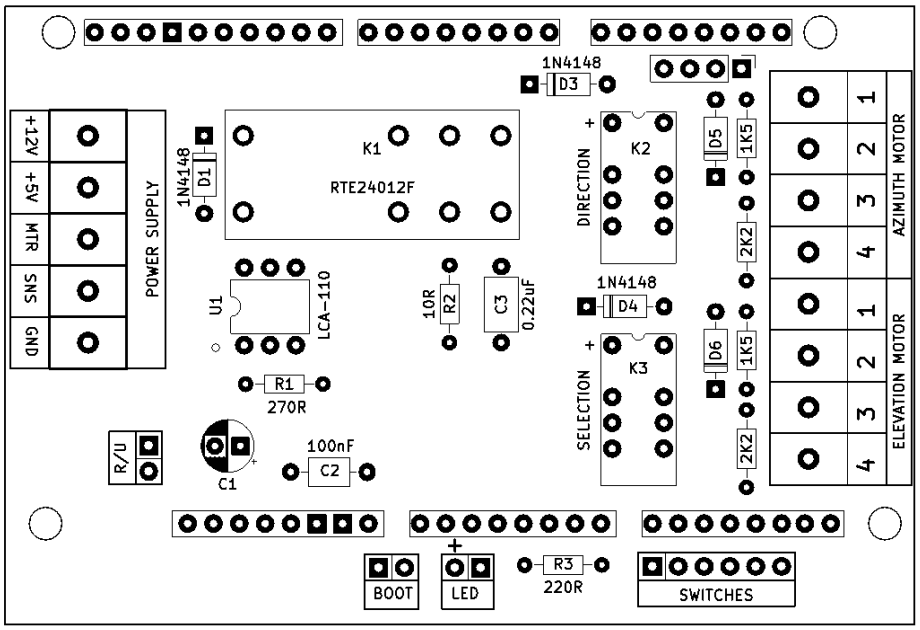
SW1 is a *Run/Update* switch. In the *Run* position, it places capacitor C2 between the processor's *RESET* line and ground. When the *RESET* input on the processor is momentarily grounded, the Arduino waits briefly to see if an attempt is being made to load new software. If it doesn't detect new code being downloaded, it simply reboots.

Unfortunately, when the processor is connected to the PC and one starts a program on the PC that interfaces with the controller, the PC program sends a momentary ground to the controller causing it to reboot. When C3 is connected (switch in the *Run* position), it swallows the pulse so no reboot occurs. The switch must be placed in the *Update* position whenever there is a need to load new software.

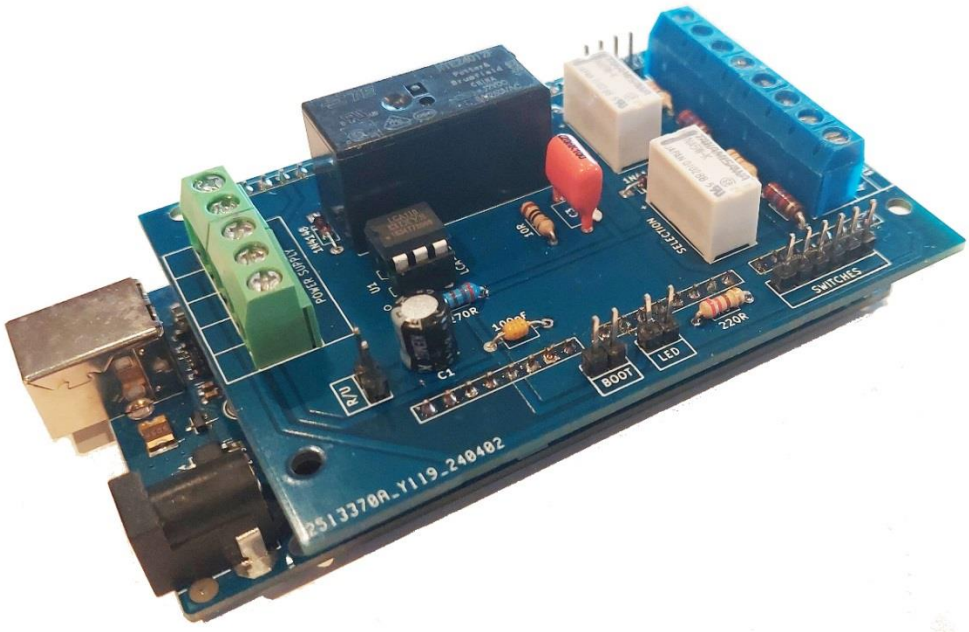
Now one might ask why J2 is labeled *BOOT* since that is normally done via the processor's *RESET* line. J2 is connected to a pushbutton on the rear of my enclosure. When that switch is pressed, it initiates a software controlled reboot, as we need to do some housekeeping before actually rebooting the processor. This is explained in the *Software* section of this document.

[1] This is what I measured with about 10 feet of cable between the controller and the rotator and using the 12V power supply for the motors. The differences between using the 12V and 15V power supplies are summarized above.

Here’s the layout for the controller PCB:



And here’s the finished PCB mounted atop the Arduino Mega:

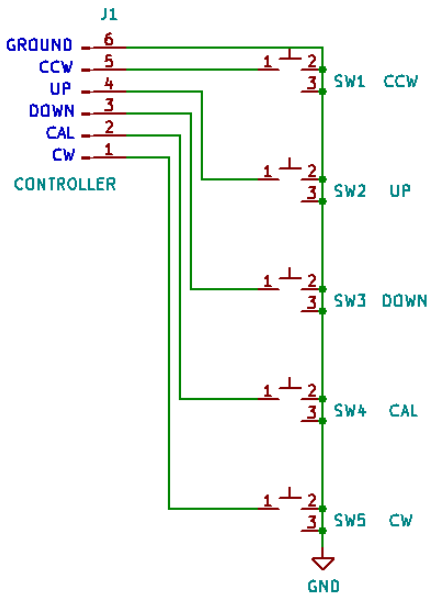


The Switch Panel

The switch panel for my build is actually made up of 2 printed circuit boards. One has the actual switches and the other is simply a nice looking front panel for the enclosure.

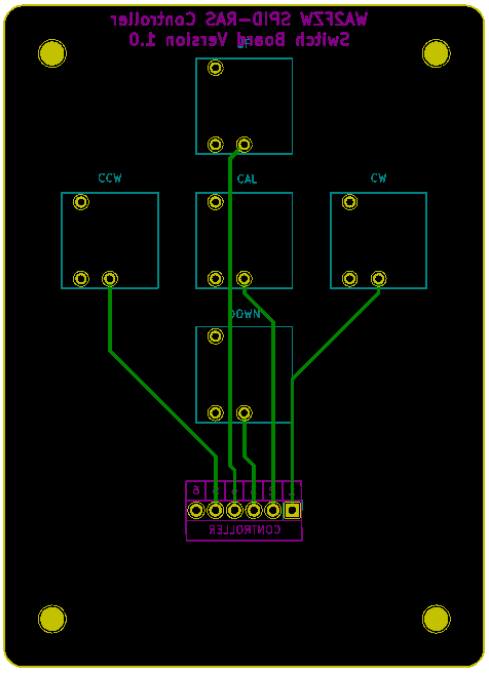
It's pretty simple; just 5 pushbuttons that ground GPIO pins on the processor which works for the Arduino as it has internal pull-up resistors that can be enabled on the GPIO pins. If one were to use a processor such as the ESP32 which does not have the internal pull-ups, those would need to be added externally.

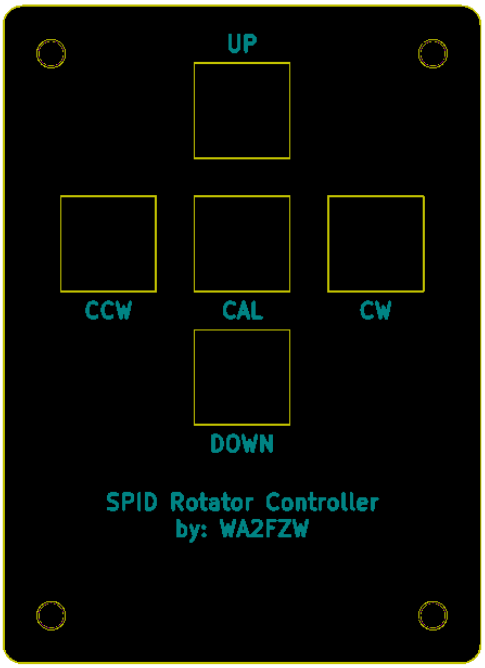
Here's the schematic:



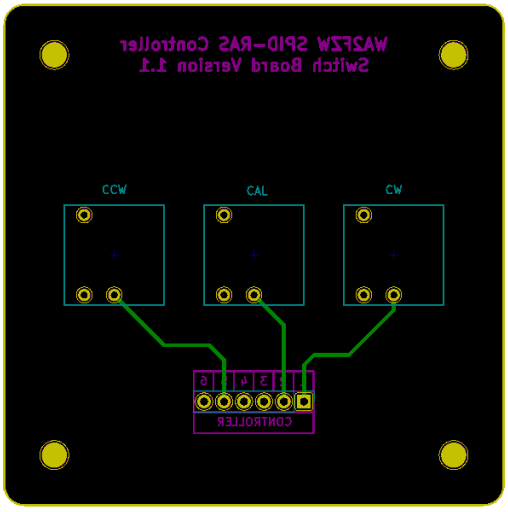
The schematic for the azimuth only version simply omits SW2 and SW3.

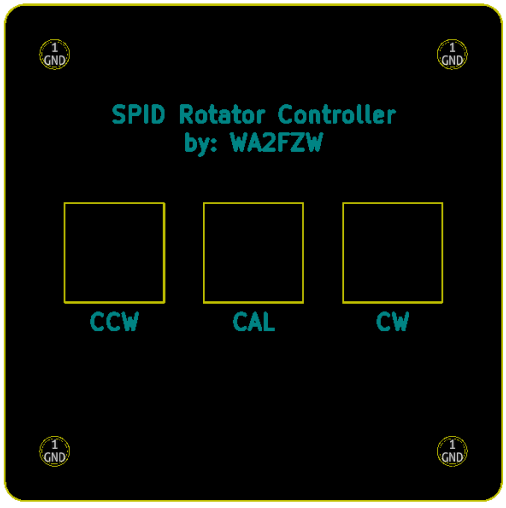
I used switches I found at the local electronics store (yes a few of those still exist) but unfortunately they came with no part numbers so you might have to use a different PCB or simply mount the pushbuttons on the enclosure, but here's what my PCB and the front panel look like for the elevation/azimuth version:





And here are the azimuth only versions:





Finished Unit

Here's front & rear panels of my unit. As noted in the Modification History, I reduced the height of the switch PCB and front panel PCB slightly from what is shown here:

Notice that I have both 4 pin Cinch type connectors and a barrier terminal strip for connections to the rotator. I used the Cinch connectors as I have a bunch of them on hand; use whatever works for you.

And here's the interior:

I would recommend a bigger enclosure! I had to make a lot of mechanical adjustments to make it all fit, for example notice the very long standoffs under the power supply PCB; without those, there was a conflict between the on/off switch and the board.

I used this particular enclosure because I already had a couple of them and there is already a panel in my equipment rack for two of these. Here are the EL/AZ and azimuth only units mounted in the rack panel:

The rack panel was custom made for me by NOVEXCOMM. Scott at Novexcomm tells me you can order the panel by specifying “Dual WA2FZW Controller Panel” as he has the drawings; he can also do one for a single controller or quad controllers. The drawings for the panel shown are included in the distribution package.

Summary of Arduino input/output pin usage

These are the pin assignments for the Arduino Mega 2560. With the exception of pins which need to be capable of handling interrupts (interrupts are no longer used however) and those with pre-assigned functions, the assignments were made in such a manner as to facilitate the PCB layout. In most cases the pin names in the software match or are similar the NET names on the schematic.

GPIO Pin #	Name	Purpose
D0	RX0	USB receive line
D1	TX0	USB transmit line
D4	SELECT	Operates the relay that selects which motor unit to run (K3)
D13	PWR_SENSE	Senses whether main power supply is turned on
D14	DIRECTION	Operates the relay that selects the direction of rotation (K2)
D18	EL_INDEX	Index pulses from the elevation motor
D19	AZ_INDEX	Index pulses from the azimuth motor
D21	LCD_SDA	Display data line
D22	LCD_SCL	Display clock line
A1	REBOOT	Initiates a software reset when LOW
A3	OPERATE	Operates the relay to apply power to the selected motor unit (K1)
A7	SAVED_LED	When lit indicates that both the current azimuth and elevation have been saved in the processor’s EEPROM. When blinking indicates that a timeout occurred
A8	CW_BTN	Pushbutton to initiate clockwise (azimuth) rotation
A9	CAL_BTN	Pushbutton to initiate a system calibration
A10	DOWN_BTN	Pushbutton to initiate down (elevation) rotation
A11	UP_BTN	Pushbutton to initiate up (elevation) rotation
A12	CCW_BTN	Pushbutton to initiate counterclockwise (azimuth) rotation

Controlling the Rotator

There are two ways one can set the azimuth and/or elevation:

- Via ASCII text commands through the Arduino's USB port
- By use of the pushbuttons

ASCII Commands

ASCII text commands can be sent at any time on the USB port.

The text commands recognized are a combination of a subset of the Yaesu GS-232 command protocol and a couple that that I made up generally for debugging purposes. The protocol is the same as that of the Yaesu GS-5500 rotator and the complete manual for that can be downloaded from the Yaesu website.

Some commands which would initiate an elevation change are disabled if the rotator is an azimuth only model.

Here is the list of the recognized commands:

Command	Action
'C' & 'C2'	The 'C' command requests the current azimuth. The 'C2' command requests both the current azimuth and elevation. Both are standard GS-232 commands.
'M' & 'W'	The 'M' command tells the azimuth rotator to move to the specified heading. The 'W' command specifies new positions for both azimuth and elevation. Both are standard GS-232 commands.
'E'	The 'E' command tells the rotator to move to the specified elevation. Note that in the standard GS-232 command set, 'E' tells the rotator to stop the elevation motor. This command is not used by any of the PC programs that I use.
'F'	The 'F' command (Calibrate) is no longer recognized (changed in Version 1.2)

‘P’	The ‘P’ (Park) command tells the rotator to move to a predetermined azimuth and elevation as defined by the ‘AZ_PARK’ and ‘EL_PARK’ symbols in the ‘SPID.h’ header file. This command is not part of the GS-232 command set.
‘Z’	The ‘Z’ (Zap) command is a testing/debugging command which sets the target and current azimuth and elevations to the specified values. The command format is the same as that of the ‘W’ command. This command is not part of the GS-232 command set.

Manual Operation

The elevation and azimuth can be changed using the pushbuttons on the front panel. The buttons have priority over commands received on the USB port. Whenever a button is operated, USB commands which would change the target azimuth or elevation are ignored. But should a command that would alter one of the targets arrive shortly after a button is released, it will be honored.

The buttons can be operated while one of the motors is running even if the movement was initiated by a command on the USB port.

When a button is operated (except the *CAL* button), the target azimuth or elevation will be incremented or decremented by 1° every *BTN_READ_TIME* milliseconds.

However, there is an option to increment or decrement the target azimuth or elevation by more than 1° when a button is depressed for longer than as specified time. The option can be enabled or disabled based on the value assigned to the *BTN_FAST* symbol (*true* or *false*).

If the *BTN_FAST* symbol is set to *true*, the value assigned to the *BTN_FAST_TIME* symbol determines how long the button must be held to start the acceleration and the value of the *BTN_FAST_INCR* symbol specifies how many degrees to increment or decrement the targets when the acceleration is in effect.

Calibration

The calibration procedure was completely overhauled in Version 1.2.

In previous versions, when the *CAL* button was pushed, the calibration procedure was initiated immediately. Now the button must be pushed and released to start the procedure.

We no longer allow calibration to be initiated via a command from the USB interface.

If the rotator is of the EL/AZ type, the elevation calibration is performed first and as in all previous versions this is done by forcing the motor to move to the zero degree endstop. If the rotator is an azimuth only type, the code for the elevation calibration is not even compiled.

If the rotator is the EL/AZ type, when the *CAL* button is pushed and released, the display will show “** EL Calibrating **” on the top line of the display and the target elevation will show 180° (it’s really set to -180°). When the elevation motor hits the lower endstop, the display will show “*** EL Timeout ***” on the top line of the display for a couple of seconds followed by “** EL Calibrated **, then “** AZ Calibrating **”.

If there is no elevation capability, upon pressing the *CAL* button, the “** AZ Calibrating **” will be displayed.

When the “** AZ Calibrating **” message appears on the top line of the display what we do is to allow the operator to adjust the current and target azimuths to match a visually observed heading of the antenna(s). The *CW* and *CCW* buttons can be used to adjust the target and current azimuths so that they match the observed azimuth. In my case, the ridge line of my roof is 042°/222° which makes it easy to verify the alignment.

When the *CAL* button is pressed and released a second time, the display will show “** AZ Calibrated **” on the top line for a couple of seconds. The calibration process is now completed.

Note that the button accelerator is not used during calibration as the assumption is that the azimuth only needs to be adjusted by a few degrees at most.

If a large adjustment needs to be made, the ‘Z’ (Zap command) can be sent on the USB connection to set the azimuth and/or elevation.

Azimuth & Elevation Save

The current azimuth and elevation are saved in the processor's EEPROM *EEPROM_TIMEOUT* milliseconds (currently set for 10 seconds) after one of the motors stops.

Both are saved whenever a timeout occurs on either motor or whenever a software reboot is initiated using the button on the rear panel.

What happens if the motor power supply is turned off but the processor is still being powered via a USB connection to the computer? I originally thought there needed to be some special handling for this situation, but it turned out to not be necessary.

If the processor is running but the motor power is turned off while one of the motors is running, a timeout will occur and the current azimuth and elevation will be saved. The saved LED will blink as usual to alert the operator; however as the display is turned off when the power supply is turned off, there will be no message displayed.

One word of caution; if the processor is not being powered via the USB port and the operator turns the power supply off before the azimuth and elevation have been saved, the EEPROM will still have the previously saved numbers and upon the next startup, the rotator most likely is out of calibration. When the controller is not connected to a computer, the operator should wait to see the saved LED illuminated before turning off the power.

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 (which I didn't end up using).

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.

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 recommend "*Beginning C for Arduino, Second Edition: Learn C Programming for the Arduino*", by Jack Purdum as a good tutorial for both the beginner and experienced programmer.

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

Arduino Libraries

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

One is for the I2C LCD display and the second is for the software reset capability. Both 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 display, you might need a different library which also might require minor changes to the software.

Initial Definitions

Definitions in the SPID.h Header File

The SPID.h header file contains a number of things that one might need to change for their particular installation.

The symbols defined in SPID.h are:

ELEVATION	When set to *false*, disables all of the elevation functions allowing the code to be used for an azimuth only rotator such as the SPID-RAU or SPID-RAK.

BAUD_RATE	The baud rate for the USB port. Note that some of the PC based programs capable of controlling the rotator require specific baud rates, so consult the manuals for the programs you intend to use to determine the proper value.
EEPROM_VALID	This is a ‘random’ number that should tell the program if the EEPROM has valid saved azimuth and elevation values.
MOTOR_TIMEOUT	This is currently set to 700 milliseconds. The rotator motors generate pulses from the reed switches approximately every 410 milliseconds[2].
MAX_AZIMUTH MIN_AZIMUTH	The maximum and minimum azimuths that are allowed
MAX_ELEVATION MIN_ELEVATION	The maximum and minimum elevations that are allowed
CAL_ADJUSTMENT	Currently set to 50 milliseconds. If the elevation motor hits one of the mechanical endstops (indicated by a timeout) we back it up for this amount of time.
DEBOUNCE	Currently set to 10 milliseconds. Accounts for contact bounce in the reed switches. Little or no contact bounce was visible on the oscilloscope when I was testing.
AZ_PARK EL_PARK	These define a specific azimuth and elevation where you can tell the antenna to go to using the text ‘P’ (park) command.
EEPROM_TIMEOUT	This specifies the amount of time to wait after both motor units have stopped before saving the current azimuth and elevation to the EEPROM. It is currently set to 10 seconds.

[2] The rotation speed and thus the pulse timing depend on the actual voltage at the motor. Long cable lengths could result in slower rotation speeds of if using the 15V regulator in the power supply, the rotation speed will be faster.

BTN_READ_TIME	The value assigned to this symbol depends on which power supply voltage is being used. 275 milliseconds works well with 12V and 200 milliseconds works well with the 15V supply. The setting *MUST* be shorter than the interval between index pulses or the motors stop and start repeatedly which is hard on the relays.
BTN_FAST BTN_FAST_INCR BTN_FAST_TIME	These enable the button accelerator. See the *Manual Operation* section for how to set these.

A note about the *MAX_ELEVATION* limit: The elevation motor is capable of moving 180° which is very useful for satellite operations however the current software has no provision to change the azimuth to the reciprocal heading when the elevation goes past 90°.

Definitions in the Main Program

There are other definitions in the main program file that are totally dependent on the hardware configuration. These should not be changed.

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 LCD object and assigning values to specific global variables prior to the *setup* function, you can't actually perform the actions necessary to put the hardware components into the desired states outside the scope of a function.

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

- Initialize the serial communications port. Note that the USB communications port can also be used for debugging using the *Serial Monitor* capability in the Arduino IDE or any other terminal emulator.

- Configure the I/O pins used to control the rotator and make sure the three relays used to select one of the motor units and direction of rotation are both off as well as the relay that provides power to the selected motor unit.
- Initialize the liquid crystal display.
- Configure the GPIO pins used to read the front panel pushbuttons and the reed switches in the motors.
- Assign some initial values to the variables used to track the movement of the azimuth and elevation motors.
- Check for valid saved current azimuth and elevation values in the EEPROM and if valid, read them. If not valid, initiate a calibration sequence then save the values.
- 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.

The *loop* function proper does the following tasks:

- Checks the status of the main power supply. If it is off, we turn the display off and any commands either from the USB connection or pushbuttons that would cause the motors to run are disabled.
- Calls the *SayHello* function which will check to see if the program name needs to be redisplayed on line 1 of the display.
- Call the *CheckButtons* and *GetCommand* functions to see if the operator is manually changing the azimuth or elevation or if there is a command from the USB interface to be processed. Note, the buttons are only looked at every *BTN_READ_TIME* milliseconds.
- If neither motor is currently running, we check to see if one or the other needs to be started. The need to start one is indicated by the fact that the current azimuth or elevation is not equal to the target azimuth or elevation.

  Note, that the elevation motor has priority over the azimuth motor. It doesn't really matter for EME operation, but this seems

to work better for satellites particularly when the azimuth needs to do a full (or almost) full 360 degree rotation.

- If one of the motors is running, we check see if the motor sent a pulse that needs to be processed and if timeout occurred. The *CheckPulse* function will stop the motors when the target azimuth or elevation is reached.
- We check to see whether or not the active the motor needs to be reversed.
- If one of the motors is running, we update the display if any of the numbers have changed.
- We see if it's been *EEPROM_TIMEOUT* milliseconds since one of the motors stopped and if so, we save the current azimuth and/or elevation in the EEPROM.
- Calls *CheckReboot* to see if the operator pushed the reset button on the back panel.

The *StartMotor* Function

StartMotor is called when one of the motors needs to be started.

It first checks to see if the motor power supply is off or if one the motors are already running. In either case, the function simply returns.

The function arguments specify which motor should be started and the direction in which it should turn.

Note that if one of the motors needs to be started, the *SELECT* and *DIRECTION* relays are first set properly, and then after a 10 millisecond delay, the *OPERATE* relay is turned on. This is done to prevent the smaller mechanical relays from being switched while power is applied to prevent contact arcing.

The *StopMotors* Function

StopMotors stops whichever motor is running. The *OPERATE* relay is turned off and after a 10 millisecond delay the *SELECT* and *DIRECTION* relays are turned off.

We also clear the *activeMotor* indicator.

The *why* argument is a *String* which will be displayed on the serial monitor if debugging is enabled.

The *CheckTimeout* Function

CheckTimeout checks to see if one of the rotator motors has timed out. That happens if one of the motors is running and we didn't see an index pulse within the time limit set by the *MOTOR_TIMEOUT* symbol (currently 700 milliseconds).

If a timeout is seen on the azimuth motor, it is assumed to be a glitch. The motor is stopped and the current azimuth is assumed to be correct. The *curAz* and *tgtAz* variables are set to the value of the *lastGoodAz* variable. The azimuth motor has no physical endstops so a positive automatic calibration is not possible.

If the elevation motor times out and the antenna was being raised, the motor is stopped and the current elevation is assumed to be correct. The *curEl* and *tgtEl* variables are set to the value of the *lastGoodEl* variable.

If the antenna was being lowered, the assumption is that it hit the endstop and the target and current elevations are set to *MIN_ELEVATION*. That might or might not actually be the case.

When a timeout occurs we put a message on the display and flash the saved LED.

The *CheckPulse* Function

CheckPulse handles the processing of a pulse from one of the reed switches. It is called everytime through the *loop* function.

The function checks to see if one of the motors is actually running and if not, it simply returns.

If one of the motors is running, it then determines which one is running and checks to see if a pulse is in progress. If the (*AZ* or *EL*) *INDEX* pin is in a *LOW* state it indicates a pulse is in progress. If the *INDEX* pin is *HIGH* it simply returns.

If the *INDEX* pin is *LOW*, we wait for the *debounce* time to expire then look to see if the pin is still in a *LOW* state. If it isn't, we assume it's a false pulse and return.

If the pin is still *LOW* after the *debounce* time has expired, then it's assumed to be a valid pulse.

Next, we wait for the *INDEX* pin to go back to the *HIGH* state, but while doing so, we also check to see that the *MOTOR_TIMEOUT* limit has not been reached. If the timeout limit is exceeded, the function returns and the *CheckTimeout* function will properly handle that situation.

Once the *INDEX* pin goes back to the *HIGH* state, we adjust the current azimuth or elevation appropriately based on the direction of movement. The above processing is the same for both motors.

The *CheckReversal* Function

The *CheckReversal* function handles the case when the operator (or a command from the PC) changes the target azimuth or elevation such that one of the motors is now turning in the wrong direction.

The logic is simple; if the motor isn't running, we do nothing. If it is running, we look to see if the current direction of rotation needs to be reversed based on a comparison of target and current azimuth or elevation. If the motor needs to be reversed, we simply stop it. The fact that the target and current azimuth or elevation are still not equal will be picked up on the next pass through the *loop* function and the motor will be restarted in the proper direction.

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

The *Calibrate* Function

The *Calibrate* function can only be invoked when the *CAL* pushbutton is operated; it cannot be initiated from a USB command.

In the case of the elevation motor, the target azimuth is set to -180° (shown as 180 on the display) which will force it to attempt to move past the 0° lower endstop limit. When the motor times out, the current and target elevations are set to the value defined by the *MIN_ELEVATION* symbol.

While the calibration process is running, either the "** AZ Calibrating **" or “**EL Calibrating **” message will appear on the top line of the display and either the “** AZ Calibrated **” or “** EL Calibrated **” messages will be displayed when each process has completed.

The procedure to actually perform the calibration is described in the *Calibration* section above.

The *CheckButtons* Function

CheckButtons checks to see if the operator has requested some manual operation by pushing one of the control buttons. It is called from the *loop* every *BTN_READ_TIME* milliseconds.

If the *activeButton* variable indicates that there was an active button the last time we checked, that button is retested to see if it is still active. If it is, the target azimuth or elevation is incremented or decremented appropriately.

The default increment or decrement is 1°, however there is an option to change the increment when a button is held for a specified amount of time; this is described in the *Manual Operation* section.

If the *CAL_BTN* is pushed, and the calibration process has not been started already, it is started when the *CAL_BTN* is released.

If no button was active on the last check, we check to see if one has gone active and take the appropriate action. We remember which one is now active in the *activeButton* variable.

The *GetCommand* Function

The *GetCommand* function checks to see if any data has arrived at the serial port.

It uses the *Serial.peek* function to see what the first character of the command is. The command character is translated to upper case, so the commands can be entered in either lower or upper case (in all cases, the command itself is a single alpha character; anything following the command character will be data.

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. Note the use of the *Serial.readStringUntil* function to read the command line. The commands are assumed to terminate with a carriage return ('\r') character.

Once the command has been read into the buffer, a *switch* statement is used to determine what the command is and take the appropriate actions.

The commands and what they do are listed in the section, *Controlling the Rotator*.

The *SayHello* Function

The *SayHello* 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 program title message needs to be re-displayed. Anytime another status message is present on the top line of the display, we leave it there for 3 seconds before re-displaying the title.

The *UpdateNumbers* Function

UpdateNumbers looks to see if the current or target azimuth or elevation has changed since it was last called, and if so, updates the display appropriately.

Note that the numbers are first converted to *String* objects using the *Int_2_String function* and then converted from a *String* to a normal *char* array. Unfortunately, the library which handles writing to the display has no provision for doing so from a *String* object.

The *CheckLCD* Function

The *CheckLCD* function is called whenever something needs to be updated on the display. It looks for data in the LCD buffers, and if anything is in one of the four, it uses the *LCDPrint* function to actually display them.

This is a leftover from the original AR-22 code. That controller used an encoder to change the azimuth and trying to update the display from within the encoder interrupt service routing (ISR) caused system lockup.

The *LCDPrint* Function

The *LCDPrint* 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 through 4, and convert those to 0 through 3 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 longer 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_String* Function

This function accepts an integer argument and creates a 3 digit *String* object with the appropriate number of leading zeroes.

The *BlinkLED* Function

This flashes the saved LED rapidly 5 times. It is currently only used when a timeout occurs or when a software reboot is initiated to get the operator's attention.

The *IncAz*, *DecAZ*, *IncEl* and *DecEl* Functions

These increment or decrement the target azimuth or elevation variables and apply the appropriate range checks. They also set the flags that the target changed and needs to be re-displayed.

The *amount* argument specifies how many degrees to increment or decrement the values.

The *CheckReboot* Function

CheckReboot looks to see if the reset button on the back panel is operated and if so, makes sure the motors are stopped, blinks the saved LED, and performs a software initiated reboot.

Known Bugs and Glitches

So far, so good!

Coming Attractions

As noted in the section describing the *loop* function, starting the elevation motor takes priority over the azimuth motor. For EME operations, it doesn't really matter as the moon moves really slow.

But for satellites, it’s a tossup! If the satellite pass is nearly overhead, giving the elevation motor priority seems best. When the satellites are lower, the azimuth motor should have priority particularly when a 360° is required.

As the azimuth motor is capable of unrestricted motion, I may allow it to turn beyond the current zero to 360° limits, or I may setup a time slicing approach which will allow moth motors to move a limited number of degrees and switch which one is operating.

In testing with SatPC32, when tracking satellites the azimuth and elevation were being saved after every small movement. I might think about adding a switch to disable the save functions whenever the operator chooses to do so.

Bill of Materials

I haven't priced everything out yet, but I'm guessing the whole thing can be built for $200 or maybe even less.

Main Controller PCB

Parts needed for the main controller circuit board:

Processor	Arduino Mega 2560
R1	270 Ohm 1/4W
R2	10 Ohm 1/4W
R3	220 Ohm 1/4W
R4 & R6	1,500 Ohm 1/4W
R5 & R7	2,200 Ohm 1/4W
C1	10uF 50V Electrolytic
C2	100nF Disk
D1, D3 & D4	1N4148
D2	General purpose LED (I used a green one) mounted on the front of the enclosure connected to the PCB by a 2-pin male header.
D5 & D6	4.7V Zener diode
K1	RTE24012F DPDT 12V Relay
K2, K3	Fujitsu NA-5W-K 5V DPDT relay
J1	5 contact screw terminal block – 0.2" spacing
J2	2-pin vertical male header – 0.1" spacing
J3	6-pin vertical male header – 0.1" spacing
J4	4-pin vertical male header – 0.1" spacing
J5, J6	4 contact screw terminal block – 0.2" spacing

SW1	SPST Toggle switch mounted on the rear of the enclosure and connected to the PCB via a 2-pin vertical male header – 0.1” spacing.
SW2	Not shown on the schematic. SPST momentary pushbutton connected to controller via J2.
U1	LCA-110 Solid state relay (6 pin DIP socket is optional).

Power Supply PCB

These parts are needed for the power supply circuit board:

R1	680 Ohm 1/4W
C1, C2, C5, C9	100uF 50V Electrolytic
C3, C4, C7, C8	100nF 25V
C6	1,000uF 50V Electrolytic
D1	GBU8A 8A Bridge rectifier
D2	1N5288
D3	1N5818
D4	General purpose LED (I used a red one) mounted on the front of the enclosure and connected to the PCB via a 2-pin vertical male header – 0.1” spacing.
L1	DLH-22-0006 100uH 100 uH inductor
U1	LM7812 Regulator
U2	LM2576-12 or LM2576-15 Regulator (see the *Power Supply* section)
U3	LM7805 Regulator
J1	2 contact screw terminal block – 0.2” spacing

J2	5 contact screw terminal block – 0.2” spacing

Switch Panel Components

J1	6-pin Male vertical header – 0.1” spacing
SW1 – SW5	Momentary SPST pushbuttons The ones that the PCB is designed for were found at the local electronics shop and have no part numbers. You might have to do something different here.

Other Components (Mounted in the Enclosure)

The following parts are not shown on the above schematics, but rather mount on front or rear panels of 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:

Enclosure	I used this one from Circuit Specialists, but as noted above, I recommend using a slightly larger one.
F1	2 amp fuse and appropriate holder
S1	Power switch
S2	Run/Update switch; SPST toggle switch
S3	Reset button; SPST pushbutton
T1	12V (or 14V) Transformer; I used this 14V one.
Display	I2C 4 line by 20 character LCD display
D1	General purpose 20mA red LED (power)

D2	General purpose 20mA green LED (azimuth saved)
Rotator Connection	You need 8 connections for the EL/AZ rotators; 4 for the azimuth only rotators. I used an 8 terminal barrier strip and 2 4 pin Cinch-Jones jacks.
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.