Wayne's Tinkering Page

NexStar™ Direct Telescope Mount Control

10/20/2015: Corrected incorrect color codes on RJ12 connector image

4/11/2016: Please see the new interface described here "NexStar™ Direct Telescope Mount Control - Revisited"

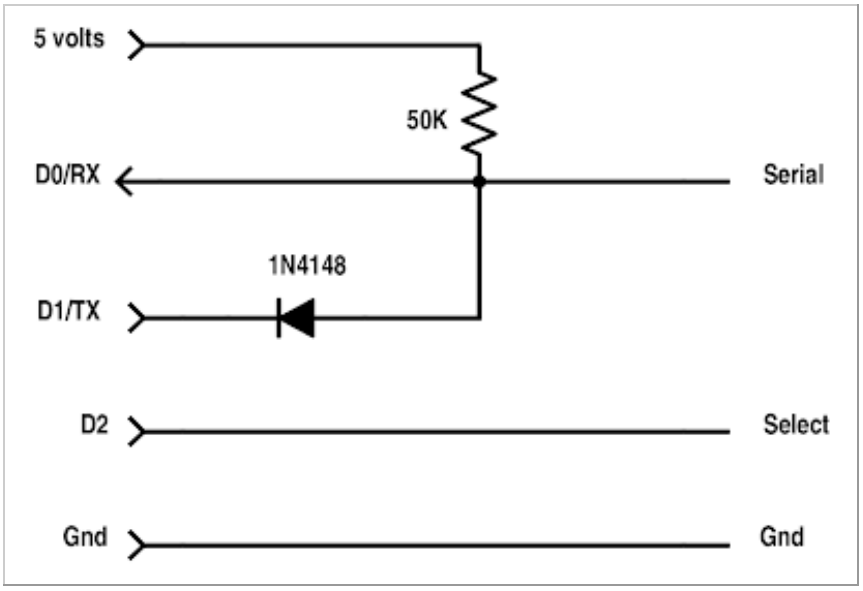
Over the summer of 2015 I got interested in communicating with Amateur Satellites (AMSATs) and the International Space Station (ISS) using the amateur 2 meter and 70 cm frequency bands. A few years back I'd gotten my Technician Class license, but had never really done much with it other than to experiment with APRS trackers. The problem with "working" amateur satellites and the ISS is that they move quickly across the sky in arc-like paths. I'd seen videos of amateurs showing how to track this path using simple, handheld yagi antennas, but I thought it would be interesting to automate this process with some type of automated tracking mechanism. I found a very interesting project by a guy who'd built his own tracker using various, off-the-shelf parts from places like ServoCity. However, on the theory that I could find some way to adapt them to use as a motorized tracker, I purchased a pair of used "NexStar" motorized telescope mounts (manufactured by Celestron for their GT-series telescopes) on eBay for a very good price. Here's new two photos of the mounts I purchased. One shows the unit attached to a photographic tripod and the other showing the interior mechanism, which consists of the motor controller PCB and a pair of DC servo motors with gear reduction and optical encoders on the main motor shaft to precisely track the position of each axis.

Click for larger view

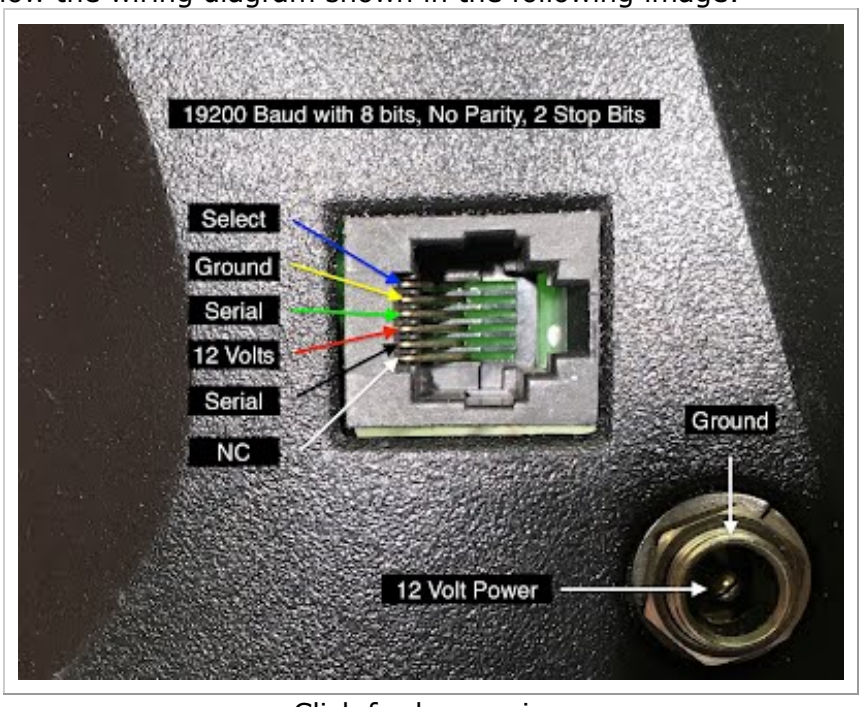
After some online research, I found various bits and pieces of documentation on these "Alt-Azimuth" type of mounts. One particularly important trove of information came from a document named "NexStar AUX Command Set" written by Andre Paquette. This was important because much of the information on controlling these motorized mounts pertains to a special set of commands that only work when sent to the handheld controller designed to attach to, and control the telescope mount. The hand controller, in turn converts these commands into a different format called the "AUX" format and sends these to the motor controller inside the mount. Not wanting to have to drag the hand controller around, too, I decided to focus on trying to unravel the AUX format so I could directly control the mount.

The AUX Protocol

After several hours of poking around and using my trusty Saleae Logic Analyzer, I managed to figure out that the AUX protocol uses serial commands sent over a single wire, bidirectional bus where both the command and response message packets traveled over a shared conductor. An additional wire serves as a SELECT signal that's used to grab control of the serial bus when it is asserted low. The SELECT line seems to be actively pulled up to 5 volts, so it only needs to be pulled low to grab control and send a command. Then, it must be released to that the motor controller in the mount can assert the same line low in order to send back the response. Here's circuit I came up with to implement this interface using an Arduino Leonardo, which I selected because it has a serial channel that's independent from the USB connection used to program the Leonardo and talk to it using the serial monitor window.



Note: it took a few hours of frustration before I finally discovered that the serial line also needs to be pulled high (50K Ohms, or greater, is required or the pull up resistor will prevent the response packets from the motor controller from being able to drive the line low.) Without this pull up, the transmitted serial commands distort and cause framing errors. I suspect this wouldn't happen if I used an active, tri-state buffer on the D1/TX line rather than the crude, diode with pull up approach. But, for simplicity, I stuck with the diode and resistor once I worked out the correct value for the resistor. To connect this circuit to the telescope mount, follow the wiring diagram shown in the following image:



Click for larger view

The 12 volts available on the 6 pin, RJ12 jack is intended to power the NexStar hand controller, but could also be used to power the Arduino Leonardo I'm using to control it. Power to the telescope mount is via the barrel jack seen in the lower right. In practice, it's designed to run from a 12 volt battery pack, but I was also able to power it using a 2 Amp, 12 Volt power supply I also got off eBay.

Caution: If you use an RJ-12 cable, be sure to select a "straight thru" cable and not a "reversed" cable. The straight thru version connects the same color wires to the same numbered pin on each end of the cable. The reversed cable swaps the wires.

Command and Response

Before I describe how the software to control this interface works, take a look at the following screen capture from my Saleae Logic Analyzer:

Click for larger view

This shows the signal timing to send a packet to the motor controller in the mount asking it to report back the version number of the software it's running. If you consult the NexStar AUX Command Set document (linked above). you'll see that packets both to and from the mount always begin with 0x3B byte, followed by a length byte, etc. You'll also notice how the Select line (top trace) goes low for the first packet, which is the command sent from the Leonard to the motor controller, then high again to signal the completion of the command. Then, a few microseconds later, it goes low again to signal the response packet coming back from motor controller.

Each packet contains a "source" byte that indicates the sender of the packet and a "destination" byte that indicates the destination. These are reversed in the response packet sent back in a response to a command. One useful aspect of the serial bus approach is that it allow additional devices to connect and share the bus. I believe Celestron uses this to add an optional GPS module to the telescope mount so it can determine its position and then adjust its celestial tracking accordingly. However, I realized that I could also use it to add my own circuitry inside the telescope mount and use the shared serial bus to communicate with it. I'd just need to assign the new device a new destination address.

Here's a breakdown of the command and response packets shown in the timing diagram above.

Command Packet for Get Version

Byte	Value	Notes
0	**0x3B**	Header byte
1	**0x03**	Packet length (excluding first two bytes and last byte)
2	**0x0D**	Packet source (0x0D indicates hand controller)
3	**0x10**	Packet destination (0x10 - Azimuth motor, 0x11 Altitude motor)
4	**0xFE**	Command (0xFE is "Get Version")
5	**0xE2**	Checksum (2's complement of bytes excluding first and last (this)

Response Packet for Get Version

Byte	Value	Notes
0	**0x3B**	Header byte
1	**0x05**	Packet length (excluding first two bytes and last byte)
2	**0x10**	Packet source (0x10 - Azimuth motor, 0x11 Altitude motor)
3	**0x0D**	Packet destination (0x0D indicates hand controller)
4	**0xFE**	Echoed Command (0xFE is "Get Version")
5	**0x05**	MS digits of version number
6	**0x15**	LS digits (after '.') of version number

7	**0xC6**	Checksum (2's complement of bytes excluding first and last (this)

And, here's an overview of the various commands that I was able to get working wth my NexStar unit and which seemed useful to my application. There are many other commands documented in Andre Paquette's PDF document (see above), but they're mostly designed for use in Astronomy and I did not explore then in any great depth. Also, while the Get Version command, 0xFE, might not seem useful beyond a one-time use, you might consider using it as a simple way to detect if the mount is connected and the interface is working properly.

Cmd	Length	Data	Function	Data Bytes Sent	Response Data
0x24	4	1	Begins Axis Move in Positive Direction	Motion speed (1-9, where 9 is fastest, 0 is stop)	Ack
0x25	1	1	Begins Axis Move in Negative Direction	Motion speed (1-9, where 9 is fastest, 0 is stop)	Ack
0xFE	3	0	Get Software Version Number	N/A	2 byte version num
0x04	6	3	Set Position of Axis (no movement)	24 bit position value (MSB first)	Ack
0x01	3	0	Get Current Position of Axis	N/A	3 bytes position (MSB first)
0x02	6	3	Goto Position Quickly	24 bit position value (MSB first)	Ack
0x17	6	3	Goto Position Slowly	24 bit position value (MSB first)	Ack

Notice that some commands send back only an "Ack" response while others send back data, such as the current position of an axis. An "Ack" packet for a command looks like the command packs except the values in the source and destination fields are reversed and the packet has a length of 4 where the data byte sent back is a "1" to indicate success. So, for example, when sending the 0x24 command (Begins axis move in positive direction) with a speed value of 9, here's are the two packets you would see on the bus:

```
3B 04 0D 11 24 09 B1   Command
3B 04 11 0D 24 01 B9   Response (Ack)
```

This specific command is used to start and stop motion on a particular axis (but more on that in a minute.) Each axis needs to be individually controlled so, for example, to slew to a new position in the sky you need to send commands to both the Azimuth (0x10 destination) and Altitude (0x11 destination) motors. Also, when first powered up, the mount has no way to determine where each axis is currently positioned. So, the first thing you have to do is to move both axes to a reference position, such as with the Azimuth pointing North and the Altitude level with the ground.

The easiest way to move the mount to a reference position is by using commands 0x24 (move positive) and 0x25 (move negative.) These commands are actually designed to work with the up/down/left/right arrow keys on the NexStar Hand Controller. So, for example, when you press the Up arrow, the controller would send the 0x24 (move positive) command with a speed value of 0x09 (max speed) and the axis would start turning in a clockwise direction (when looking at the inside of the mount where the telescope clamp is visible.) The axis would continue to turn until you release the Up arrow, at which time the controller would send the 0x24

command again, but with a speed value of 0. Likewise, when the down arrow is pressed, the controller would send the 0x25 (move negative) command. When using these commands, it's important to remember that the axis will continue to move until it receives a stop command (speed = 0.)

Once you've moved both axes to the reference position, you can use command 0x04 (Set Position) on both axes to give this position a value. Note: you can make this position any 24 bit value you desire, but it makes more sense to make it the zero position, in my opinion. In effect, this command reset the motor controller's internal count it uses to track where the axis is currently pointing and set it, instead, to the value you send in the data field of command 0x04 (Set Position.) You can use command 0x01 (Get Position) to read the current position of an axis at any point in time, even when the axis is in motion. However, it is recommend that you do not do this too quickly, as repeating the command too quickly might confuse the controller.

Using one of the Goto commands, 0x02 (fast movement), or 0x17 (slow movement) you can move either axis to any position by sending out a 24 bit value and the axis will move to this position and then stop. For example, if the Altitude axis is currently positioned at 0x000000, sending a fast goto 0x100000 command will cause that axis to rotate clockwise (again, as viewed from the side where the telescope attaches) by 1/16th of 360 degrees, or 22.5 degrees. On my unit, the Altitude axis takes about 8 seconds to make this move in fast mode, which means the axis moves at approximately 2.8 degrees/second, which is just a bit less than 1/2 the speed the second hand moves on a clock. The Goto Position Slowly (0x17) command, in comparison, takes about 45 seconds to move the same distance, or roughly 1/2 degree/second.

If you send the 0x01 (Get Current Position) command after making an axis move, you'll observe that the position you get back does not exactly match the position you requested the axis to move to. For example, after moving in fast mode from 0x100000 back to 0x000000, Get Position reported 0x0003F3. Making the same move in slow mode, Get Position read back exactly 0x000000, so you might consider using slow mode for greater positional accuracy. However, 0x0003F3 is equivalent to an inaccuracy of .0217 degrees, so the positional accuracy in fast mode is actually not too bad. And, I did not make repeated measurements in slow mode to determine if it can hit this position so accurately each time.

Code for the Leonardo (ATMega32U4)

Next, I'll present a simple Arduino sketch you can use to test the commands described above. As shown in the interface schematic (above), the sketch uses only 3 pins; D0(RX), D1(TX) and D2. Pin D2 is used as the SELECT signal and the code takes advantage of the Arduino'd ability to dynamically convert pins back and forth from input pins to output pins and uses this to emulate an open-collector pull down mechanism on this pin. It does this by setting the pin LOW and then flipping the pin between being an Input Pin, or an Output Pin. When set as an Output, the LOW state of the pin pulls the SELECT line LOW. When flipped to an Input Pin, the SELECT line is pulled up to a high state by the pull up resistor in the motor control board. The code then also tests the state of the SELECT pin to sense when the motor control board is sending back a response packet. If you want to watch the raw packet exchange, set `#define DEBUG` to 1, otherwise leave it set to 0.

```
#define DEBUG 0

#define AZM 0x10
#define ALT 0x11
```

```
uint8_t   maxSpd = 0x09;
uint8_t   stopSpd = 0x00;
uint8_t   posDir = 0;
uint8_t   negDir = 1;
uint8_t   rsp[10];
uint32_t  position[2];
uint8_t   selAxis = ALT;

void setup() {
  Serial.begin(115200);
  Serial1.begin(19200, SERIAL_8N2);
  pinMode(2, INPUT);
  digitalWrite(2, LOW);
}

void receive () {
  while (Serial1.available()) {
    unsigned char cc = Serial1.read();
#if DEBUG
    if (cc < 0x10)
      Serial.print('0');
    Serial.print(cc, 16);
    Serial.print(' ');
#endif
  }
}

void sendCmd (uint8_t dst, uint8_t id, uint8_t* data, uint8_t
len) {
    pinMode(2, OUTPUT);
    delayMicroseconds(50);
    Serial1.write(0x3B);
    receive();
    Serial1.write(0x03 + len);
    receive();
    Serial1.write(0x0D);
    receive();
    Serial1.write(dst);
    receive();
    Serial1.write(id);
    receive();
    uint8_t crc = (0x03 + 0x0D) + len + dst + id;
    for (uint8_t ii = 0; ii < len; ii++) {
      Serial1.write(data[ii]);
      receive();
      crc += data[ii];
    }
    Serial1.write(((~crc) + 1) & 0xFF);
    receive();
    Serial1.flush();
    receive();
    while (digitalRead(2) == LOW) {
      pinMode(2, INPUT);
    }
#if DEBUG
    Serial.println();
#endif
    for (int ii = 0; ii < 1000; ii++) {
      if (digitalRead(2) == LOW)
```

```
        break;
      delayMicroseconds(50);
    }
    int idx = 0;
    while (digitalRead(2) == LOW) {
      delayMicroseconds(50);
      if (Serial1.available()) {
        unsigned char cc = Serial1.read();
        rsp[idx++] = cc;
      }
    }
#if DEBUG
    for (int ii = 0; ii < idx; ii++) {
      unsigned char cc =rsp[ii];
      if (cc < 0x10)
        Serial.print('0');
      Serial.print(cc, 16);
      Serial.print(' ');
    }
    Serial.println();
#endif
}

// Move to position at maximum motor speed (9)
void gotoFast (uint8_t dst, uint32_t pos) {
  uint8_t tmp[] = {(pos >> 16) & 0xFF, (pos >> 8) & 0xFF, pos &
0xFF};
  sendCmd(dst, 0x02, (uint8_t*) &tmp, 3);
}

// Move to position at slow motor speed (?)
void gotoSlow (uint8_t dst, uint32_t pos) {
  uint8_t tmp[] = {(pos >> 16) & 0xFF, (pos >> 8) & 0xFF, pos &
0xFF};
  sendCmd(dst, 0x17, (uint8_t*) &tmp, 3);
}

// Set position of axis to 'pos' (does not move axis)
void setPosition (uint8_t dst, uint32_t pos) {
  uint8_t tmp[] = {(pos >> 16) & 0xFF, (pos >> 8) & 0xFF, pos &
0xFF};
  sendCmd(dst, 0x04, (uint8_t*) &tmp, 3);
}

uint32_t getPosition (uint8_t dst) {
  sendCmd(ALT, 0x01, 0, 0);
  return (rsp[5] << 16) + (rsp[6] << 8) + rsp[7];
}

void loop() {
  receive();
  if (Serial.available()) {
    unsigned char cc = Serial.read();
#if DEBUG
    Serial.println();
    Serial.println((char) cc);
#endif
    switch (cc) {
      case '[':
```

```
            selAxis = ALT;
            break;
          case ']':
            selAxis = AZM;
            break;
          case '0':
            // Stop Slew of Selected Axis
            sendCmd(selAxis, 0x24, &stopSpd, 1);
            break;
          case '1':
            // Slew Selected Axis in negative direction
            sendCmd(selAxis, 0x25, &maxSpd, 1);
            break;
          case '2':
            // Slew Selected Axis in positive direction
            sendCmd(selAxis, 0x24, &maxSpd, 1);
            break;
          case '-':
            // Rotate in negative direction by 0x100000
            gotoFast(selAxis, position[selAxis & 1] =
    (position[selAxis & 1] - 0x100000) & 0xFFFFFF);
            break;
          case '+':
            // Rotate in positive direction by 0x100000
            gotoFast(selAxis, position[selAxis & 1] =
    (position[selAxis & 1] + 0x100000) & 0xFFFFFF);
            break;
          case 'x':
             // Move Selected Axis to position 0x000000
            gotoFast(selAxis, 0x000000);
            break;
          case 'g':
            // Get Selected Axis current position
            Serial.println(getPosition(selAxis) & 0xFFFFFF, 16);
            break;
          case 's':
            // Set Selected Axis current position to 0x000000
            setPosition(selAxis, 0x000000);
            position[selAxis & 1] = 0x000000;
            break;
          case 'v':
            // Get AZM Version number
            sendCmd(AZM, 0xFE, 0, 0);
            Serial.print(rsp[5]);
            Serial.print('.');
            Serial.println(rsp[6]);
            break;
        }
        // Delay before processing next command to avoid overrun
        delay(500);
      }
    }
```

To use the code, open the Arduino's Serial Monitor window and use the text input line at the top to send single character commands to the unit. As a first step, select an Axis to control by sending either '[' to select the Alt Axis, or ']' to select the Azm Axis. The choice you make will stay selected until you restart the program. Then, use the '1', '2', or '0' commands to move the selected Axis in the negative direction ('1'), the positive direction ('2'), or stop ('0'). Use these

commands to move both Axes to a reference position from which yo can issue other commands. After each Axis is moved into the correct position, use the 's' command to set this position as 0x000000 for that Axis. From this point, you can try out some of the additional commands describe below:

```
[    Select the Alt Axis
]    Select the Azm Axis
0    Stop motion on the Selected Axis
1    Start motion in the Negative Direction on the Selected Axis
2    Start motion in the Positive Direction on the Selected Axis
s    Set the Current Position of the Selected Axis to 0x000000
g    Get the Current Position of the Selected Axis
x    Move the Selected Axis back to position 0x000000
+    Move Selected Axis in the Positive Direction by 0x100000 (1/16 of a revolution)
-    Move Selected Axis in the Negative Direction by 0x100000 (1/16 of a revolution)
v    Get Version Number for the Selected Axis (should return the same result for
either Axis)
```

Note: you should be able to combine multiple commands on one line, such as typing "[x]x" to move both Axes back to the zero position at the same time. However, this functionality has not been tested extensively, so caveat emptor. You can also download the Arduino sketch using the link provided below.

NexStarTest2.zip (1k) Wayne Holder, ... v.1