Arxrobot Firmware

Table of Contents

Getting Started

You can download the `arxrobot_firmware` from the Arxterra GitHub releases page: https://github.com/arxterra/arxrobot-firmware/releases.

Near the top of the `arxrobot_firmware` tab you will see a sequence of definitions.

```
arxrobot_firmware §   battery_selector   command   fuel_gauge   pinouts_robot.h   sparkfun_TB6612FNG   telemetry
#include "pinouts_robot.h"     // replace with your pinout

#define Rosco FALSE
#define PaperBot TRUE
#define bluetooth TRUE     // Leonardo class rovers - Serial used for USB, Seriall for bluetooth (i.e., USART)
#define pinger FALSE       // FALSE if no ultrasonic sensor is used.
#define debug  FALSE
```

Figure 0.1   Configuring the code for a PaperBot robot

Based on the Arxterra robot you have and the features included, set these fields accordingly. The setting for Figure 0.1 correspond to a standard PaperBot. Figure 0.2 show the settings for a mini-Rosco rover. The mini-Rosco uses bluetooth communictions versus a standard Rosco which uses USB OTG, and in this example the mini-Rosco is equipped with an ultrasonic pinger.

```
#define Rosco TRUE
#define PaperBot FALSE
#define bluetooth TRUE     // Leonardo class rovers - Serial used for USB, Seriall for bluetooth (i.e., USART)
#define pinger TRUE        // FALSE if no ultrasonic sensor is used.
#define debug  FALSE
```

Figure 0.2   Configuring the code for a mini-Rosco

Once you have configured your code. Open the `pinouts_robot.h` tab and make sure your rover's pinouts are correct. You can update the table as needed.

```
arxrobot_firmware §   battery_selector   command   fuel_gauge   pinouts_robot.h §

//Motor A
const int PWMA = A3;  // Speed control
const int AIN1 = 9;   // Direction
const int AIN2 = 8;   // Direction

//Motor B
const int PWMB = 5;   // Speed control
const int BIN1 = 7;   // Direction
const int BIN2 = 4;   // Direction
```

Figure 0.3   Defining Arduino-to-robot pinouts

The remainder of this document covers how to modify the `arxrobot_firmware` for your robot.

Commands

Command and telemetry data are sent as byte arrays formatted into packets. The format of a command packet is shown in Figure 1.0. Each command packet consists of the following bytes.

1. The Packet ID is an Arxterra unique byte indicating the start of a packet. For commands, this byte by definition is hexadecimal A5.
2. The Packet Length is the number of bytes following minus the LRC byte.
3. The Command ID is an 8-bit code identifying the instruction to be executed.
4. Depending on the instruction, the command ID byte is followed by some number of bytes.
5. The Longitudinal Redundancy Check (LRC) byte provides a simple parity check of the command packet. The LRC is generated over the entire packet.
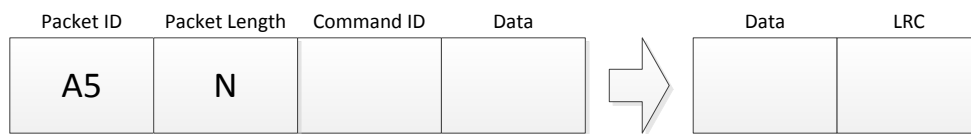
Packet ID	Packet Length	Command ID	Data		Data	LRC
A5	N					

Figure 1.0   Command Packet Format

Move (0x01) Forward Half Speed Test Example

In this example the robot is commanded to proceed forward at half speed (50% duty cycle).

```
i =  0  1  2  3  4  5  6  7
N =        1  2  3  4  5
     A5 05 01 01 80 01 80 A1    MOV forward half speed


         sendPacket
data[0] A5 = Command Packet ID
data[1] 05 = Packet Length N w/o parity byte
data[2] 01 = Move ID
data[3] 01 = Left Motor Forward
data[4] 80 = Half Speed (128/255)
data[5] 01 = Right Motor Forward
data[6] 80 = Half Speed (128/255)
data[7] A1 = Longitudinal Redundancy Check (LRC)
```

Robot Commands

The latest table of command definitions is included in the `pinouts_robot.h` file. Within the Arduino IDE these command definitions may be found under the `pinouts_robot.h` tab. Figure 2.0 shows the command definitions for arxrobot-firmware alpha version 0.8.2.

```
// Arxterra Commands to Numeric Value Mapping
//              Data[0] =    CMD TYPE | Qual | Arguments
//                           bit  7654321   0    Bytes    N = 1 + bytes
#define MOVE            0x01    // 0000000   1    4
#define CAMERA_MOVE     0x02    // 0000001   0    4
#define CAMERA_HOME     0x04    // 0000010   0    0
#define CAMERA_RESET    0x05    // 0000010   1    0
#define READ_EEPROM     0x06    // 0000011   0    3
#define WRITE_EEPROM    0x07    // 0000011   1    3 + b
#define SAFE_ROVER      0x08    // 0000100   0    0
#define SLEEP           0x0A    // 0000101   0    0
#define WAKEUP          0x0B    // 0000101   1    0
#define HEADLIGHT_OFF   0x0C    // 0000110   0    0
#define HEADLIGHT_ON    0x0D    // 0000110   1    0
#define PING_INTERVAL   0x10    // 0001000   0
#define PING            0x11    // 0001000   1    1
#define HEADING         0x12    // 0001001   0
#define CURRENT_COORD   0x13    // 0001001   1         (float) Latitude, (float) Longitude
#define WAYPOINT_COORD  0x14    // 0001010   0
#define WAYPOINT_DELETE 0x16    // 0001011   0
#define WAYPOINT_MOVE   0x17    // 0001011   1
#define WAYPOINTS_OFF   0x18    // 0001100   0
#define WAYPOINTS_ON    0x19    // 0001100   1
#define WAYPOINTS_CLEAR 0x1A    // 0001101   0
```

Figure 2.0   Commands to Numeric Value Map

Each command is comprised of a 7-bit command type field, followed by a 1-bit qualifier. The qualifier allows 1-byte commands with a single Boolean argument. For example Headlight ON/OFF (`0b0000110:1/0`).

Custom Commands

The Arxterra control panel allows you to create your own custom commands based on the unique requirements of your robot. Figure 3.0 shows a control panel designed to support a Hexapod robot.

Figure 3.0   Custom Control panel for a Hexapod Robot

Control Panel User interface (UI) design widgets include radio buttons, sliders, drop down menus, and more.

The Arxterra Arduino program reserves command IDs **0x40** to **0x5F** for implementing these user defined commands on the robot.

To learn how to create your own custom UI widget, you can view Jeff Gome's video series [here](#) or read "[Creating Custom Commands With the Arxterra Application](#)" by Tommy Sanchez.

Implementing Commands

How the Arxrobot firmware detects, reads, and handles commands

All Arduino programs contain a `setup()` and a `loop()` method. Commands are detected in the `loop()` method.

```
void loop(){

  #if bluetooth
  if(Serial1.available() ) commandDecoder();
  #else
  if(Serial.available() ) commandDecoder();
  #endif
```

Once detected, commands are decoded in the `commandDecoder()` function. You can find the command decoder code under the command tab within the Arduino IDE (Integrated Development Environment).

arxrobot_firmware	battery_selector	command	fuel_gauge	pinouts_robot.h	sparkfun_TB6612FNG	telemetry

```
/* Public Function */

void commandDecoder()
{
```

The commandDecoder method implements a FSM which reads the incoming byte array (see Figure 3.0).
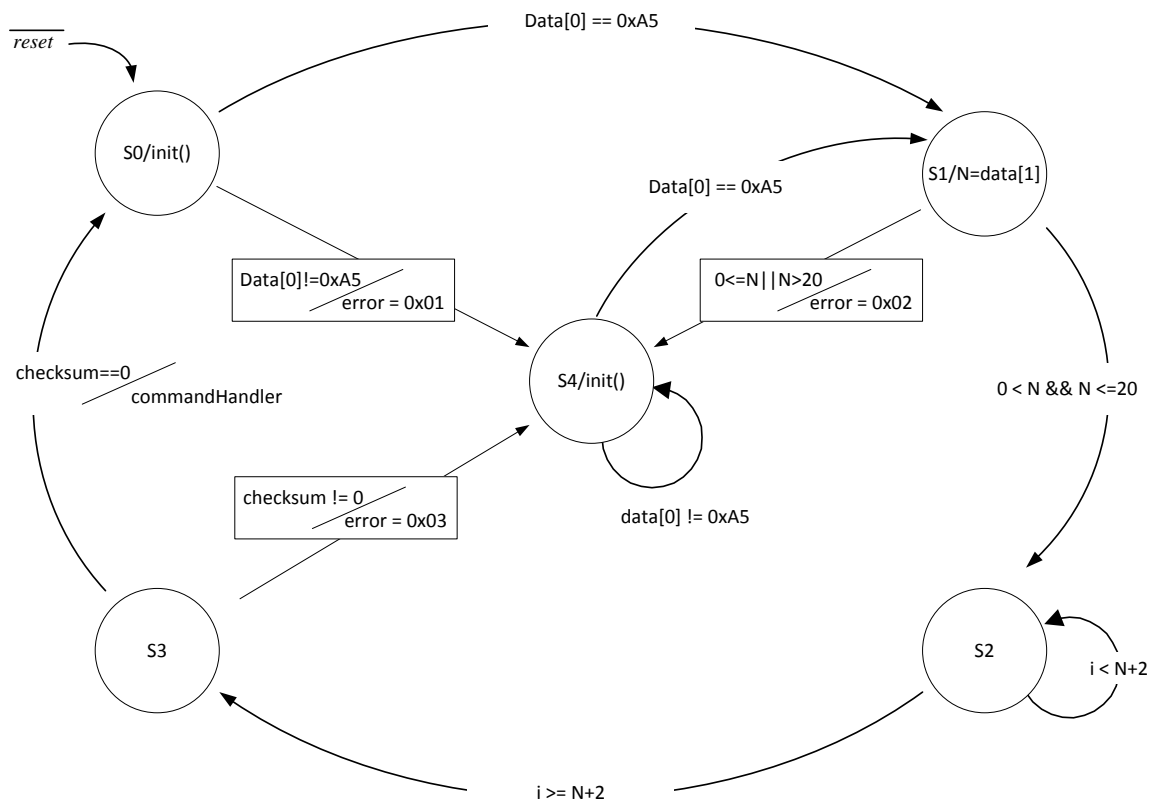
Figure 3.0: Command Decoder Finite State Machine (FSM)

If the command is successfully decoded then the `commandHandler` function is called from state 3 of the FSM.

```
case 3:  // Checksum
if (checksum == 0){
  commandHandler(data, N); // Go To function where you run robot code based on Command Received.
```

The `commandHandler` function definition includes parameters `data` and `N`. The `data` parameter points to the received byte-array. The N parameter is the length of the packet as previously defined (see Figure 1 and associated definitions). It is simplest to understand these two parameters by again looking at the `move` (0x01) command example.

```
i =     0  1  2  3  4  5  6  7
N =        1  2  3  4  5
```

```
Data = A5 05 01 01 80 01 80 A1    MOV forward half speed
```

The first byte A5 identifies this as the start of an Arxterra command packet. The second byte 05 is the value sent as the N argument. The third byte (`data[2]`) is the Arxterra command, in this case `move` (0x01). One of the first things the command handler function does is to assign this third byte to the unsigned byte variable cmd.

```
uint8_t cmd = data[2];
```

Shortly following this assignment statement is a series of if-else if statements which allow direct action to be taken by the Arduino based on the command received.

```
        if (cmd == MOVE) {
          ...
        }
        else if (cmd == CAMERA_MOVE){
          ...
        }
        else if (cmd == CAMERA_RESET){ ...
```

The next section will look at how to adapt the existing code, within each command, to the specific needs of your robot.

How to integrate the Arduino code for your robot's unique command(s)

IMPORTANT

Before you integrate your robot's unique functions into the `arxrobot` application, begin by developing and testing your new functions in a separate Arduino `ino` file(s). Once you know they work you can add them to the `arxrobot_firmware` code, the subject of this section

---

You can take two approaches to modifying the code for your robot. The simplest is to throw away the code within the `else if` block and adding your own. The second is to add your code in such a way that the original functionality of the code is not lost. Here is how to accomplish the second option.

Going back to the `arxrobot_firmware` tab (see Introductions Section) you will see a sequence of definitions.

```
#define PaperBot TRUE
#define Rosco FALSE
#define Pathfinder FALSE

#define bluetooth TRUE
#define pinger FALSE            // FALSE if no ultrasonic sensor is used.
#define debug  TRUE
```

The `#define` C++ preprocessor directive or in this case more accurately a one-line macro definition, tells the compiler to replace any occurrence of the identifier (for example Rosco) with the text provided on that line. For example, whenever the compiler sees the word Rosco it will replace it with FALSE

(which in turn is defined as 0). Add a new line for your robot. Declare it `TRUE` and the remaining robots `FALSE`. For example

```
#define BiPed TRUE
#define PaperBot FALSE
```

 In a similar fashion add any sensors or actuators unique to your robot.

Next, open the command tab and scroll down to the command block you are implementing. The following example assumes you are modifying the `MOVE` command.

```
if (cmd == MOVE) {

  /***********************************
  * motion command = 0x01
  * motordata[3]   left run    (FORWARD = index 1, BACKWARD = index 2, BRAKE = index 3, RELEASE = index 4)
  * motordata[4]   left speed  0 - 255
  * motordata[5]   right run   (FORWARD, BACKWARD, BRAKE, RELEASE)
  * motordata[6]   right speed 0 - 255
  * example
  * forward half speed  0x01, 0x01, 0x80, 0x01, 0x80 0101800180
  ***********************************/

  #if Rosco == TRUE
    // Adafruit L293D motorshiled
    move_L293D(data);
  #elif Pathfinder
    // Pololu VNH5019 motorshield (Pathfinder)
    move_VNH5019(data);
  #elif PaperBot
    // Sparkfun TB6612FNG
    move_TB6612FNG(data);
  #endif
}
```

Append a `#elif` preprocessor conditional directive (before `#endif`). Within this new conditional block, add a function call to your unique command implementation.

```
#elif BiPed
  walk_BiPed(data);
```
Save and close the Arduino IDE. Add the `ino` file(s) containing the functions that are unique to your robot.

Reopen the Arduino IDE. The new ino file(s) will now appear as tab(s). Remove any `setup()` or `loop()` methods from your development software.

Compile your code and verify that there are no errors.

Unpacking/Formatting Command Arguments
As mentioned earlier, all command data is in the form of a byte array, formatted as an Arxterra command packet. Your unique commands may want data formatted as a type other than an 8-bit byte. In this example, four (4) bytes are converted into a 32-bit "unsigned long" data type and saved in variable `ping_interval`.

```
ping_interval = (((unsigned long)data[3]) << 24) | (((unsigned long)data[4]) << 16) | (((unsigned long)data[5]) << 8) | (unsigned long)data[6];
```

Command Exception Codes

01   Start byte 0xA5 expected
02   Packet length out of range 1 - 20
03   LRC checksum error
04   Undefined command decoder FSM state
05   Array out of range i >= 23

Telemetry

The latest table of telemetry ID definitions is included in the `pinouts_robot.h` file. Within the Arduino IDE these telemetry ID definitions may be found under the `pinouts_robot.h` tab. Figure 4.0 shows the telemetry ID definitions for arxrobot-firmware alpha version 0.8.2.

```
/******************** Robot Telemetry ****************
 * source: Arxterra
 ***************************************************/
 // Telemetry Identifiers to Numeric Value Mapping
#define   MOTOR1_CURRENT_ID   0x01            // motor 1 is left motor
#define   MOTOR2_CURRENT_ID   0x02            // motor 2 is right motor
#define   TEMP_SENSOR_ID      0x03            // temperature sensor
#define   RANGE_LEFT_ID       0x04            // ultrasonic range 1 is left
#define   RANGE_RIGHT_ID      0x05            // ultrasonic range 2 is right
#define   CLEAN_BATTERY_ID    0x06            // Arduino battery
#define   DIRTY_BATTERY_ID    0x07            // DC and servo motors battery
#define   PAN_POSITION_ID     0x08            // originally defined as pan and tilt
#define   TILT_POSITION_ID    0x09            // not in original definition
#define   EEPROM_RESPONSE_ID  0x0A            // sent in response to EEPROM Read Command
#define   EMERGENCY_ID        0x0B

#define   COMMAND_ECHO_ID     0x0D
#define   EXCEPTION_ID        0x0E            // arduino error code exception (see subtable below)
#define   PONG_ID             0x11
#define   ROUTE_STATUS_ID     0x12
#define   WAYPOINT_ARRIVE_ID  0x13

 /******************** 0x0E Exception Codes ****************
 *    01     Start byte 0xA5 expected
 *    02     Packet length out of range 1 - 20
 *    03     LRC checksum error
 *    04     Undefined command decoder FSM state
 *    05     Array out of range i >= 23
 ***********************************************************/

 /******************** Emergency Codes ********************
 *    0x0100    Watchdog timeout 0x01 with ancillary byte 0x00
 ***********************************************************/
#define WATCHDOG_TIMEOUT    0x0100
```

Command and telemetry data are sent as byte arrays formatted into packets. The format of a telemetry packet is shown in Figure 4.0. Each telemetry packet consists of the following bytes.

1. The Packet ID is an Arxterra unique byte indicating the start of a packet. For telemetry, this byte by definition is hexadecimal CA.
2. The Packet Length is the number of bytes following minus the LRC byte.
3. The Telemetry ID is an 8-bit code identifying the sensor data contained in the packet.
4. Depending on the sensor, the telemetry ID byte is followed by some number of bytes.
5. The Longitudinal Redundancy Check (LRC) byte provides a simple parity check of the telemetry packet. The LRC is generated over the entire packet.
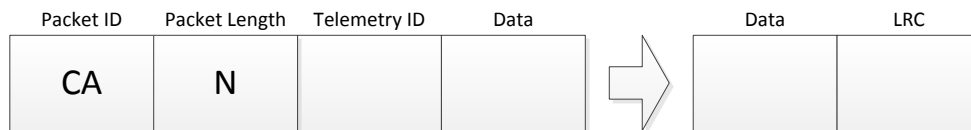
Figure 4.0   Telemetry Packet Format

Calculating the LRC byte, Telemetry Example

One trick you can use to calculate the LRC byte for a command packet is to send the command with the LRC byte set to zero.

First, turn on the debug mode. Open the `arxrobot_firmware` tab and scroll to the configuration block. Turn on the debug mode, while turning off both Bluetooth and the watchdog timer.

```
/**************** Robot Configuration **************
 * Set Rover, Pinouts, Connection Type, and Pinger *
 ***************************************************/
#include "pinouts_robot.h"    // replace with your pinout

#define Rosco FALSE
#define PaperBot TRUE
#define bluetooth FALSE       // Leonardo class rovers - Serial used for USB, Seriall for bluetooth (i.e., USART)
                              //   Set bluetooth to TRUE when using CoolTerm
#define pinger FALSE          // FALSE if no ultrasonic sensor is used.
#define debug  TRUE
#define watchdog FALSE        // turn on/off watchdog timer
```

Next, send the command packet with the LRC field set to zero[1]. In this example I will send the camera home command. To learn how to send test commands and display telemetry, read the "Testing Robot Commands" section of this document.
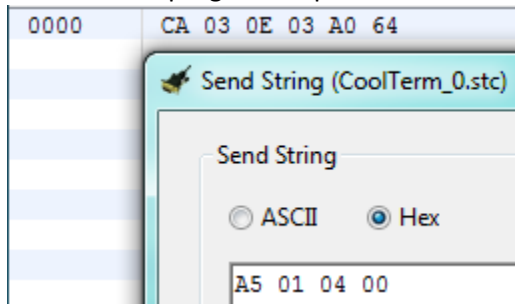
```
; command = camera home w/ incorrect parity
A5 01 04 00

Command Packet ID A5
N = 1
Camera Home = 4
```

---

[1] In this example I am using the CoolTerm application.

```
Parity 00 = should be A0
```

The arxrobot program responds with telemetry packet **CA 03 0E 03 A0 64**

```
0000       CA 03 0E 03 A0 64
```

Send String (CoolTerm_0.stc)

Send String

○ ASCII   ● Hex

```
A5 01 04 00
```

```
data[0] CA = Packet ID
data[1] 03 = Packet Length N+1
data[2] 0E = Command Exception ID = telemetry id
data[3] 03 = LRC Checksum Error   = data[0] = high byte of word sent
data[4] A0 = Expected Checksum     = data[1] = low byte  of word sent
data[5] 64 = Packet Checksum
```

The "Telemetry ID" is **0E** which is sent by the `commandDecoder()` function, indicating an error was detected when trying to decode the command. As seen in the Figure below, the checksum error **03** is "thrown" from state 3 of the FSM, where the LRC checksum byte is checked (see "Command Exception Code" section of this document for other error codes). This is the first byte (data[0]) of a 16-bit data word returned by the program. Data ordering is big-endian. The second data byte (data[1]) contains the LRC byte expected by the command decoder, in this example **A0**.

```
case 3:  // Checksum
if (checksum == 0){
  commandHandler(data, N); // Go To function where you run robot code based on Command Received.
  i = 0;
  checksum = 0;
  next_state = 0;
}
else{
  throw_error(word(0x03,checksum));
  i = 0;
  checksum = 0;
  next_state = 4;  // exception
}
break;
```

We now know the correct LRC byte to send with this command (0xA0).

```
; command = camera home w/ correct parity
A5 01 04 A0
```

Assuming you are in **debug mode**, the arxrobot program responds with telemetry packet **CA 02 0D 04 C1**
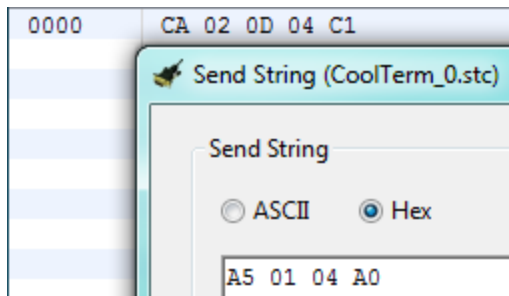
```
data[0] CA = Telemetry Packet ID
data[1] 02 = Packet Length N+1      sendWordPacket
data[2] 0D = Command Echo
data[3] 04 = Camera Home
data[4] A0 = Packet Checksum
```

Which lets us know the command was received without any errors.

Testing Robot Commands

It is often desirable to test commands locally before testing remotely. This removes the potential problems introduced by adding the smart device (Android Phone), internet cloud, and Arxterra control panel layers. To test locally you will need a terminal application program. The following instructions use the CoolTerm program.

1. Plug in your Arduino (Uno, Leonardo, Mega ADK, …).
2. Within the Arduino IDE, under Tools, set the Board Type and Serial Port.
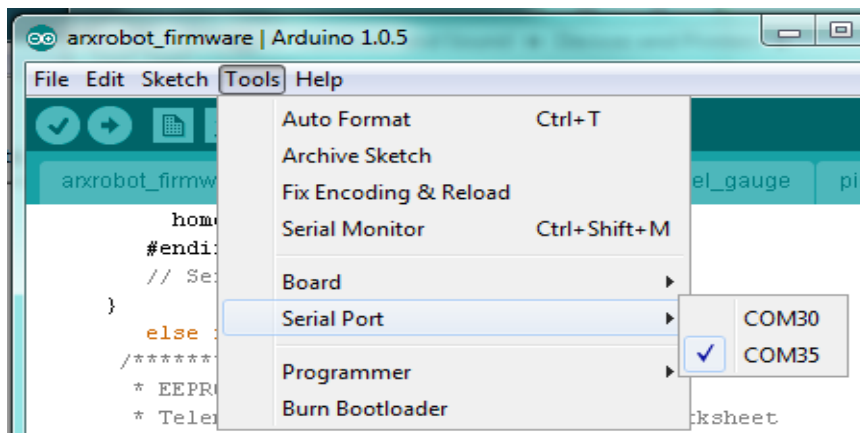
3. If you are not sure what serial port is used by the Arduino, open "Devices and Printers" window from the Start Menu. Locate and left-click the Arduino icon. Select properties from the pop-up menu. From the properties menu select the Hardware tab and locate the serial port. In the Figure below, communications (serial) port 35.
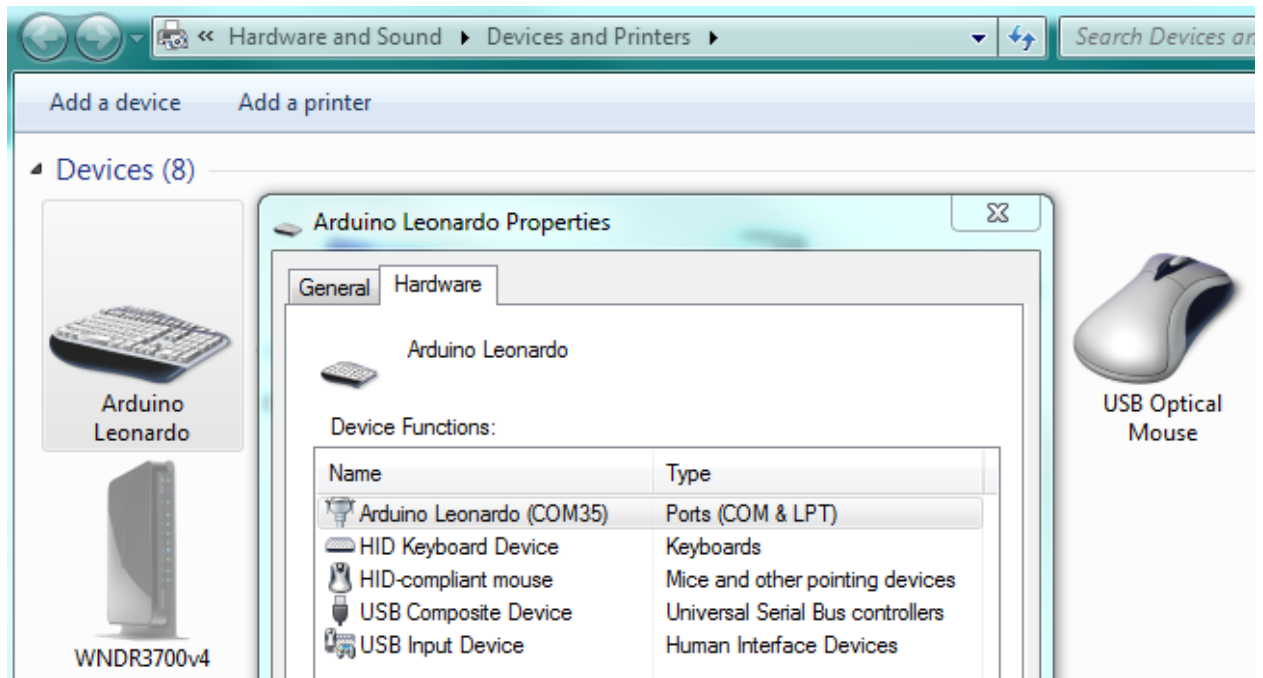
Troubleshooting

The Arduino Leonardo appears as a keyboard to your device and does not work on all USB port types (1.1, 2.0, and 3.0). If the Leonardo is recognized (computer beeps) but driver shows up as not installed, try moving cable to another port. If the Leonardo is not recognized (computer does not beep), the USB may provide power only or is broken. Try another USB cable.

4. Upload the `arxrobot_firmware`.
5. Launch CoolTerm
6. Click View ASCII
7. Click Options Icon and Serial Port. Set serial port (see step 3) and baud rate. Your baud rate setting should match the Serial baud rate set by your Arduino script. If you are not sure, open the `arxrobot_firmware` tab and scroll down to `setup()` method.

```
void setup(){
  Serial.begin(9600);      // legacy rovers operated at 57600
  #if bluetooth
  Serial1.begin(9600);
  #endif
```
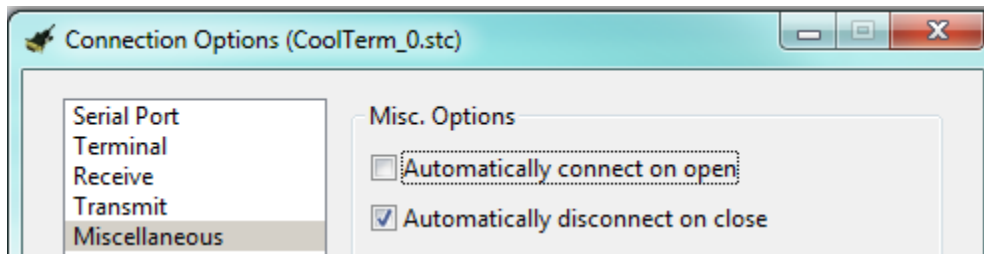
Here is how my Leonardo based configuration looks.
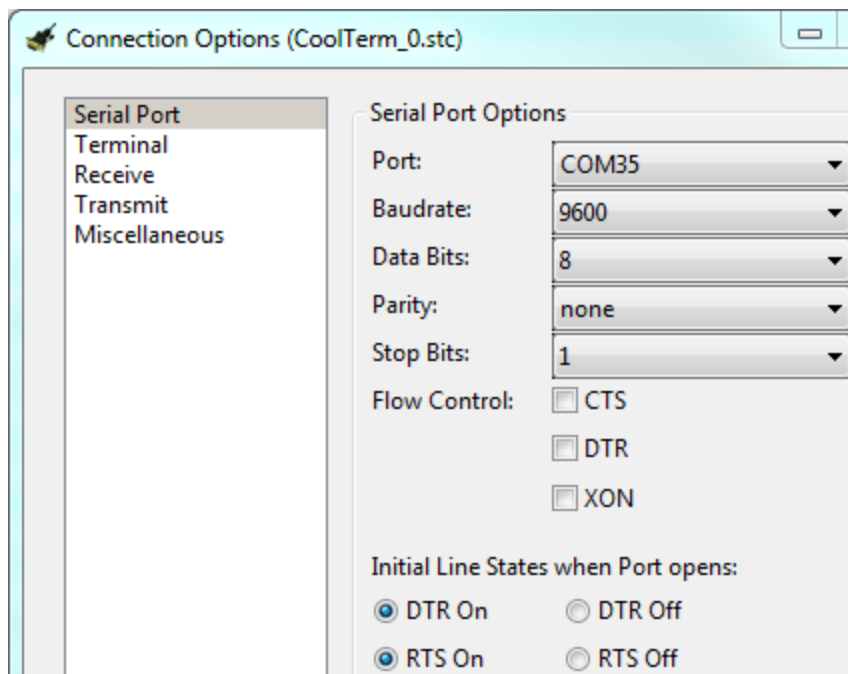
Verify following Terminal settings. Line Mode selected with Local Echo.
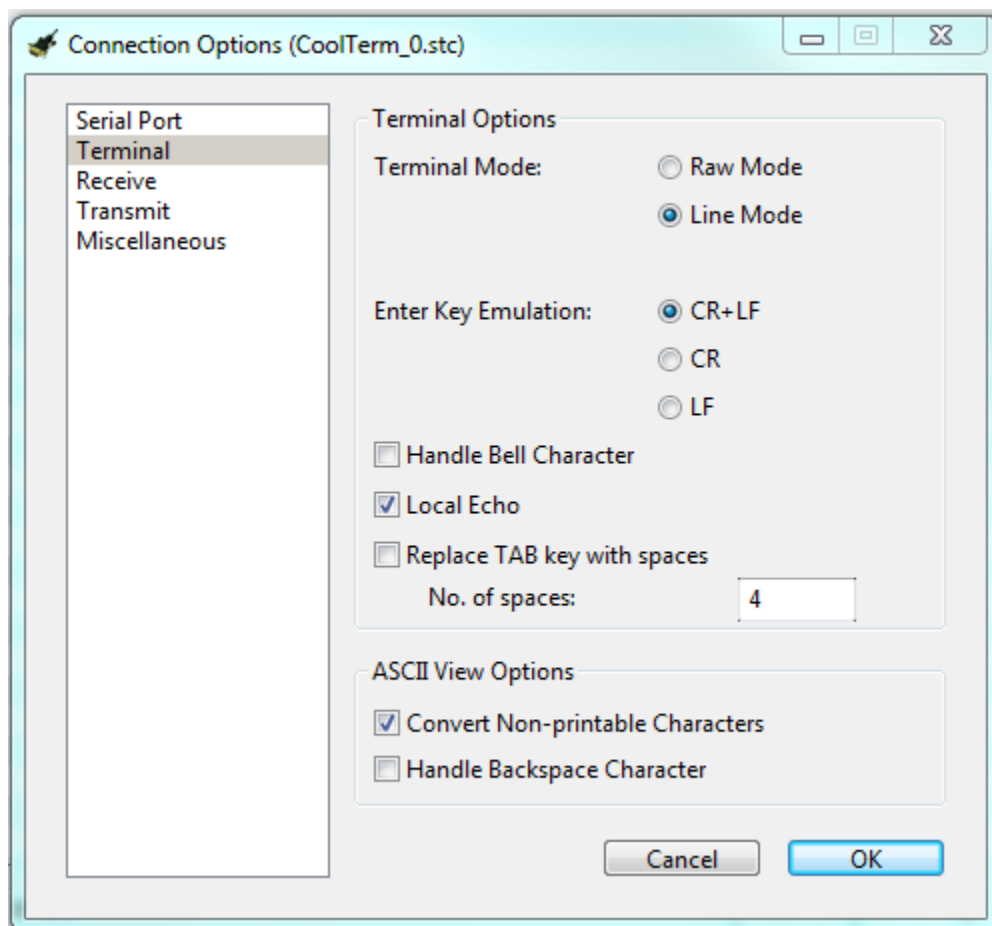
If you like, you can set application to close connection when the CoolTerm application is closed.
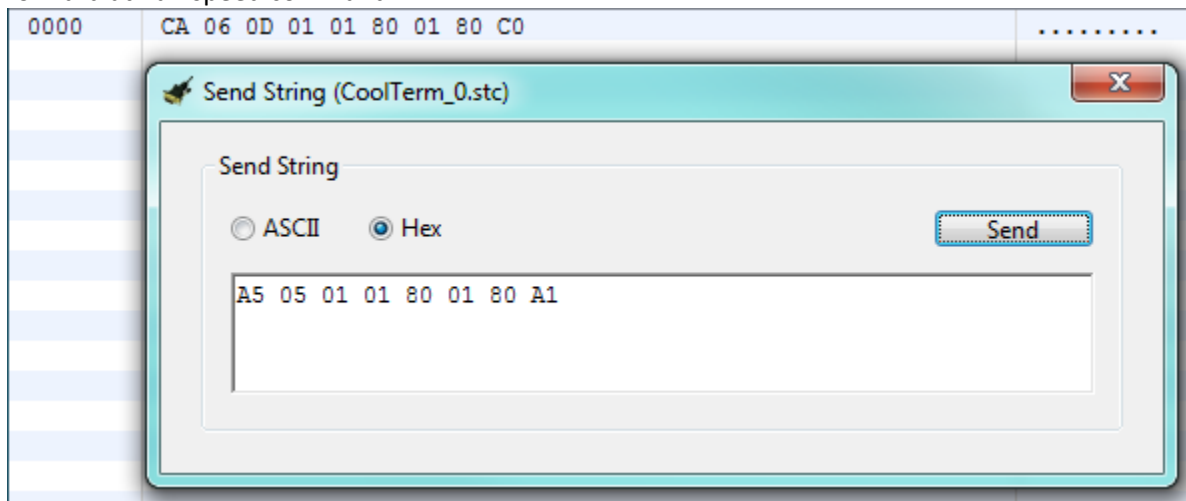

Leave remaining windows at default settings.

8. To save time in the future, **Save** your setting and **Open** the next time you work on your application.

9. Open, **Connect** the serial port communications between CoolTerm and the Arduino.

To send a command, open the Send String window from the "Connection" item on the menu bar or press CTRL-T, select Hex, and enter the command string. In the example below, I am sending the move forward at half speed command.

Mapping pins

ATmega2560-Arduino http://arduino.cc/en/Hacking/PinMapping2560
ATmega328-Arduino http://arduino.cc/en/Hacking/PinMapping168

Set Serial Monitor Baud Rate to match communication standard (`usb = 57600, Bluetooth v2 = 9600`)

To Do List

1. Pingers code to stop rover included but not tested/debugged.
2. Need to write code for pan/tilt offset angle.
3. Continue to clean up code.

Servo Notes

- The min pulse width (default is 544 microseconds) corresponds to the minimum angle (0-degree), shaft rotates fully left.
- The max pulse width, (defaults to 2400 microseconds) corresponds to The maximum angle (180-degree), shaft rotates fully right - rotation is left-to-right clockwise.

ATmega internal temperature sensor

```
#define ATMEGA_TEMP    8
http://forum.arduino.cc/index.php/topic,38043.0.html
http://forum.arduino.cc/index.php/topic,26299.0.html
```
Problem is time required when you switch between voltage reference sources for ADC to stabilize. This would limit how often we check this internal sensor.

ArxRover needs to add cell phone temperature.

Saved in UserState File on Android Phone

```
public var cameraAdjustForMotion:Boolean = false;
public var cameraCanPan:Boolean = true;
public var cameraCanTilt:Boolean = true;
public var cameraConfigDefault:CameraConfig;
public var cameraConfigMotion:CameraConfig;
public var cameraIndex:int;
public var roverName:String;
```

Defaults set in the Control Panel's Control Options pop-up Window

```
Duty Cycle Steps      6 (4 -12)
Polling msec          500 (300 - 800)
Minimum Duty Cycle    100 (10 - 140)    *** ArxRover ***
Top Duty Cycle        212 (180 - 255)
Motion State Display  ON  (Motion Text Display)
Control Tips Display   ON  (Tool Tips)
Range Sensor Display  ON  (Numeric and Graphical Icon of
Ultrasonic Data)
```

While Loops

Waiting in a while loop is always dangerous you may want to consider adding a loop count exception or check for memory available. http://playground.arduino.cc/code/AvailableMemory

Appendix A   Test Cases

Move (0x01) Forward Half Speed Test

```
i =  0  1  2  3  4  5  6  7
N =        1  2  3  4  5
     A5 05 01 01 80 01 80 A1    MOV forward half speed


        sendPacket
data[0] A5 = Command Packet ID
data[1] 05 = Packet Length N w/o parity byte
data[2] 01 = Move ID
data[3] 01 = Left Motor Forward
data[4] 80 = Half Speed (128/255)
data[5] 01 = Right Motor Forward
data[6] 80 = Half Speed (128/255)
data[7] A1 = Longitudinal Redundancy Check (LRC)
```

Camera Home (0x04) Test

```
command = camera home w/ incorrect parity
A5 01 04 00

Command Packet ID A5
N = 1
Camera Home = 4
Parity 00 = should be A0

telemtry
CA 03 0E 03 A0 64
        sendPacket
data[0] CA = Telemetry Packet ID
data[1] 04 = Packet Length N+1     sendWordPacket
data[2] 0E = Exception ID           = id

for(uint8_t i = 0; i < N=2; i++)

data[3] 03 = LRC Checksum Error  = data[0] = high byte of word sent
data[4] A0 = Expected Checksum   = byte[1] = low byte  of word sent
data[5] 64 = Packet Checksum
                                 N = 2  <= length argument

command = camera home w/ correct parity
A5 01 04 A0

telemetry
```

```
CA 02 0D 04 C1
data[0] CA = Telemetry Packet ID
data[1] 04 = Packet Length N+1      sendWordPacket
data[2] 0D = Command Echo
data[3] 04 = Camera Home
data[4] C1 = Packet Checksum
```

EEPROM Write (0x07) Test

Write Command

A5 14 07 00 00 10 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 00

```
Start Byte A5
N = 20 or 0x14
EEPROM Write Command  07
EEPROM Address 0x0000 00 00
Number of Bytes 16    10
Data  01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10
Parity Byte 00
```

Appendix B   Arduino MEGA ADK Support

Arxterra code traces its origins to the Pathfinder rover which used an Arduino ADK and communicated over the Android Device Bridge (ADB), which is no longer supported by Google. Current Arxterra class robots use USB OTG, Bluetooth v2, and in the future Bluetooth LE for communications. Therefore for readability, legacy ADB code elements have been deleted. The last arxrobot version with support for the USB ADB is paperbot_v6.

How to Update Adb Library to Support Arduino version 1.0+

The Adb library has the following modifications made to bring it into compliance Arduino version 1.04 (Compiles without errors)

1. Adb.h, usb.cpp, and max3421e.cpp
2. Change the line: #include "wiring.h" to #include "Arduino.h"
3. Runing on ADK generates a "OSCOKIRQ failed to assert" error. See
   http://forum.arduino.cc/index.php?topic=68205.0

If using the ADB Library found in the arxterra Github then these changes have already been made.