Going Beyond StandardFirmata - Adding New Device Support

by **MrYsLab** on March 26, 2014

Table of Contents

Intro: Going Beyond StandardFirmata - Adding New Device Support

INTRODUCTION

Have you ever thought of using Firmata for your Arduino project, only to find out that the device that you wish to use is not supported by Firmata? This article demonstrates the step by step approach I used to add stepper motor support to PyMata and to its associated Arduino Sketch, FirmataPlus. After reading this tutorial you should be ready to extend Firmata for your device.

Before We Begin - Some Background Information About Arduino/Firmata

So what is Firmata? Quoting from the Firmata web page, "Firmata is a generic protocol for communicating with microcontrollers from software on a host computer."

Arduino Firmata uses a serial interface to transport both command and report information between an Arduino microcontroller and a PC, typically using a serial/USB link set to a rate of 57600 bps. The data transferred across this link is binary in nature and the protocol is implemented in a **client/server** model.

The **server** side is uploaded to an Arduino microcontroller in the form of an Arduino Sketch. The StandardFirmata Sketch, included with the Arduino IDE, controls the Arduino I/O pins, as commanded by the client, and it reports input pin changes, as well as other report information back to the client. FirmataPlus is an extended version of StandardFIrmata.

The Arduino **client** used for this article is PyMata. It is a Python application that is executed on a PC. It both sends commands to, and receives reports from, the Arduino server.

Why Use Firmata?

Arduino microcontrollers are wonderful little devices, but processor and memory resources are somewhat limited. For applications that are processor or memory intensive, there is often little choice but to offload the resource demand onto a PC in order for the application to be successful.

But that's not the only reason for using StandardFirmata. When developing lighter weight Arduino applications, a PC can provide tools and debugging capabilities not directly available on an Arduino microcontroller. Using a "fixed" client and server helps isolate variability to a PC application, where it is more easily managed. Once the application is perfected, it can be translated to a custom, standalone Arduino Sketch.

Why Use PyMata?

At the time of this writing, PyMata is the **only** Python client that implements the complete StandardFirmata protocol. It is a multi-platform library that supports Windows, MAC, and Linux operating systems right out of the box!

In addition, PyMata was designed so that users could easily extended it to support additional sensors and actuators, not currently supported by StandardFirmata.

Not long after PyMata was initially published, it was extended to support the Arduino Tone Library, and to support up to six simultaneous HC-SR04 type sonar distance sensors. The server side extensions for these features are provided in FirmataPlus.

Overview of the PyMata Client and FirmataPlus Server

PyMata is a high-performance multi-threaded Python application. Its "command thread" translates user API calls into Firmata protocol messages and forwards these messages to the Arduino microcontroller. The "reporter thread" receives, interprets and acts upon the Firmata messages received from the Arduino microcontroller.

A customized version of StandardFirmata, called FirmataPlus, is included with the PyMata distribution. It currently adds the following to StandardFirmata:

- A Piezo device interface using the Arduino Tone Library
- Support for multiple ultrasonic distance sensors using the NewPing library
- Limited support (Arduino Uno only) for rotary encoders using the adafruit rotary encoder library
- A debug print function to print internal Sketch values to the Python console.

Understanding Firmata Data Representation

Firmata uses a serial communications interface to transport data to and from the Arduino. The Firmata communications protocol is derived from the MIDI protocol which uses one or more 7 bit bytes to represent data. Because 7 bits can hold a maximum value of 128, a data item that has a value greater than 128, must be "disassembled" into multiple 7 bit byte chunks before being marshaled across the data link. By convention, the least significant byte (LSB) of a data item is sent first, followed by increasingly significant components of the data item. The the most significant byte (MSB) of the data item is the last data item sent.

So for example, if a parameter called **max_distance** with a decimal value of **525** needs to be sent across the serial link, it first needs to be "disassembled" into 7 bit chunks.

Here is a discussion of how this is accomplished:

The value 525 decimal is equivalent to the hexadecimal value of 0x20D, a 2 byte value. To get the LSB, we mask the value by AND'ing it with 0x7F. Both "C" and Python implementations are shown below:

```
// "C" implementation to isolate LSB
int max_distance_LSB = max_distance &0x7f ; // mask the lower byte

# Python implementation to isolate LSB
max_distance_LSB = max_distance & 0x7F # mask the lower byte
```

After masking, max_distance_LSB will contain 0x0d. 0x20D & 0x7F = **0x0D**

Next, we need to isolate the MSB for this 2 byte value. To do this, we will shift the value of 0x20D to the right, 7 places.

```
// "C" implementation to isolate MSB of 2 byte value
int max_distance_MSB = max_distance >> 7 ; // shift the high order byte

# Python implementation to isoloate MSB of 2 byte value
max_distance_MSB = max_distance >> 7 # shift to get the upper byte
```

After shifting, max_distance_MSB will contain a value of **0x04**.

When the "chunkified" marshaled data is received, it needs to be reassembled into a single value. Here is how the data is reassembled in both "C" and Python

```
// "C" implementation to reassemble the 2 byte,
//     7 bit values into a single value
int max_distance = argv[0] + (argv[1] << 7) ;
```

```
# Python implementation to reassemble the 2 byte,
#    7 bit values into a single value
max_distance = data[0] + (data[1] << 7)
```

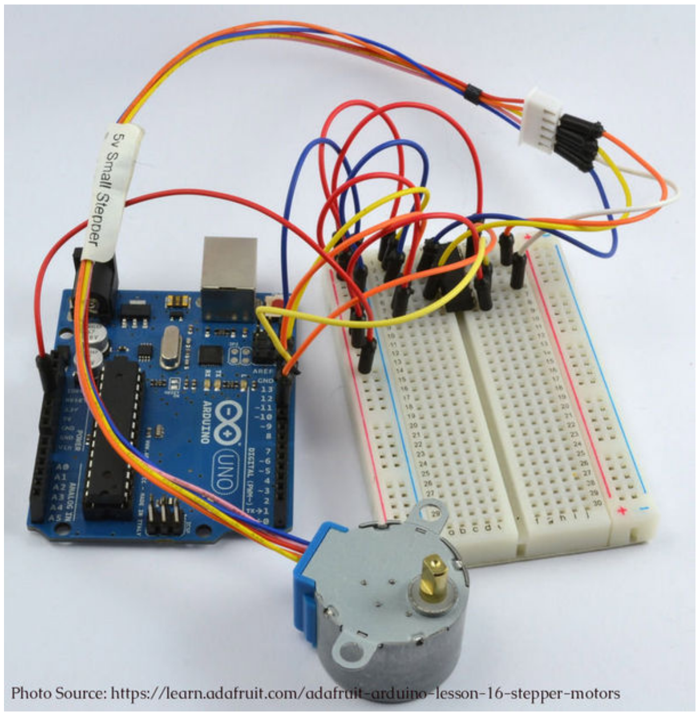
After reassembly, the value once again equals 525 decimal or 0x20D hexadecimal.

This disassembly/reassembly processes may be performed by either the client or server.

PyMata

A Python Client Abstraction Layer for the

Arduino Firmata Protocol


5v Small Stepper
Photo Source: https://learn.adafruit.com/adafruit-arduino-lesson-16-stepper-motors

Step 1: Before Writing Any Code, Analyze the Device Library Public Interface

Before creating or modifying any PyMata or FirmataPlus code to support a device, it is essential to understand how the device and its libraries function. For the stepper motor, the Simon Monk/adafruit tutorial provides excellent insight into using stepper motors and using the Arduino Stepper Library.

Let's look at how the Arduino Stepper library's public interface is defined in Stepper.h.

Stepper.h contains constructors to support both 2 conductor and 4 conductor motors. For each motor, the user specifies the number of steps per motor revolution, and the Arduino digital pin numbers that will control the motor.

```
class Stepper {
 public:
   // constructors:
   Stepper(int number_of_steps, int motor_pin_1,
                                int motor_pin_2);


   Stepper(int number_of_steps, int motor_pin_1,
                                int motor_pin_2,
                                int motor_pin_3,
                                int motor_pin_4);
```

There is also a public method to set the speed of the motor:

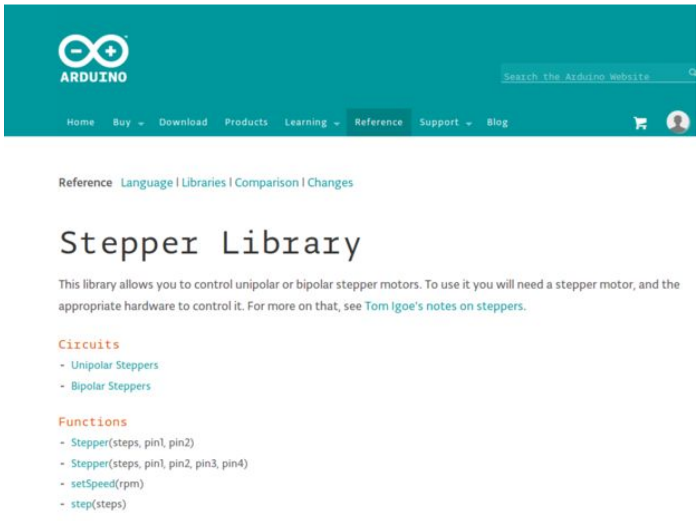
```
   // speed setter method:
   void setSpeed(long whatSpeed);
```

A third public method specifies how to move the motor a given number of steps. Implied in this method is that a positive number moves the motor in one direction, and a negative number moves the motor in the opposite direction. This can be determined either by reading the implementation source code or by experimenting with the hardware.

```
   // mover method:
   void step(int number_of_steps);
```

The final public method retrieves the library's version number:

```
int version(void);
```



Reference Language | Libraries | Comparison | Changes

Stepper Library

This library allows you to control unipolar or bipolar stepper motors. To use it you will need a stepper motor, and the appropriate hardware to control it. For more on that, see Tom Igoe's notes on steppers.

Circuits
- Unipolar Steppers
- Bipolar Steppers

Functions
- Stepper(steps, pin1, pin2)
- Stepper(steps, pin1, pin2, pin3, pin4)
- setSpeed(rpm)
- step(steps)

Step 2: Specifying and Designing The New PyMata API and Reporter Method Interfaces

Using the Stepper.h public interface as a guide, we can begin to design the PyMata stepper motor API and reporter method interfaces.

First, we need a method that will instruct the serial library to instantiate a motor object. Let's call this method, **stepper_config**, and similar to the stepper library's constructors, we will allow the caller to specify the number of steps per revolution, as well as to provide a list of motor control pins. This list, at the callers option, will contain 2 pin values for a 2 conductor motor or 4 values for a 4 conductor motor.

Here is the signature for the API method **stepper_config**.

```
def stepper_config(self, steps_per_revolution, stepper_pins):
   """
   Configure stepper motor prior to operation.
   @param steps_per_revolution: number of steps per motor revolution
   @param stepper_pins: a list of control pin numbers - either 4 or 2
   """
```

Referring to Stepper.h, the library provides separate methods to set the speed of the motor and to move the motor a given number of steps.

Our PyMata interface will need to do the same, but we are going to make a design decision and combine both operations into one method.

We will name this method **stepper_step** and its API signature is shown below.

```
def stepper_step(self, motor_speed, number_of_steps):
   """
   Move a stepper motor for the number of steps at the specified speed
   @param motor_speed: 21 bits of data to set motor speed
   @param number_of_steps: 14 bits for number of steps & direction
                           positive is forward, negative is reverse
   """
```

We also need to provide a method to allow the user to request the Stepper Motor Library version. Because of the asynchronous nature of Firmata reporting, we will need to provide separate methods to send the version request to the Arduino, and to retrieve the reply. Both of these methods are part of the public API.

The request method will have the following API signature:

```
def stepper_request_library_version(self):
   """
   Request the stepper library version from the Arduino.
   To retrieve the version after this command is called, call
   get_stepper_version().
   """
```

Before discussing the final API method, we need to specify the signature of the reporter method that processes the data sent from the Arduino to the client. This data is in the form of two 7 bit bytes that need to be reassembled to form the library version number that will be stored by the client.

```
def stepper_version_response(self, data):
   """
   This method handles a stepper library version message sent
   from the Arduino and stores the value until the user requests
   its value using get_stepper_version()
   @param: data -Two 7 bit bytes that contain the library version number
   """
```

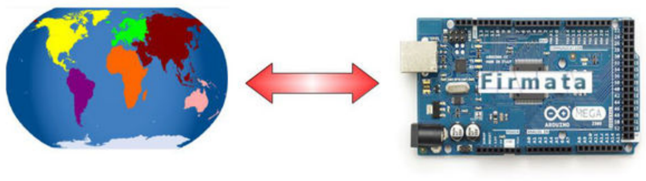
The value returned by the Arduino in the stepper_version_response message will be stored internally by PyMata. To retrieve the value, the user calls the get_stepper_version API method. To allow for Arduino data transfer and processing times, a maximum wait time parameter is specified. The default value for this parameter is 20 seconds. If the Arduino replies before the maximum time expires, the value will be returned immediately and the timer is canceled. The signature for this method is:

```
def get_stepper_version(self, timeout=20):
   """
   @param timeout: specify a max time to allow arduino to process
                   and return the version
   @return: the stepper version number if it was set.
   """
```

An Important Note

All PyMata "command" methods are implemented in **pymata.py**. Therefore when we code **stepper_config, stepper_step, stepper_request_library_version and get_stepper_version,** they will all be added to pymata.py.

All "reporter messages" originating from the Arduino are handled by **pymata_command_handler.py.** Therefore the implementation of **stepper_version_response** will be added to that file.



Step 3: Selecting and Specifying The SysEx Command and Pin Mode

Firmata was designed to be user extensible. The mechanism that provides this extensibility is the **Sys**tem **Ex**clusive (SysEx) messaging protocol.

The format of a SysEx message as defined by the Firmata Protocol, is shown in the illustration above. It begins with a START_SYSEX byte with a fixed value of hexadecimal 0xF0, and is followed by a unique SysEx command byte. The value of the command byte must be in the range of hexadecimal 0x00-0x7F. The command byte is then followed by an unspecified number of 7 bit data bytes and finally, the message is terminated with an END_SYSEX byte, that has a fixed value of hexadecimal 0xF7.

After receiving a complete SysEx command from a client, the server dispatches the command to the Sketch's sysexCallback() function, where it is processed.

```
// Excerpt From FirmataPlus.ino


/*==================================================================
* SYSEX-BASED commands
*================================================================*/
void sysexCallback(byte command, byte argc, byte *argv){
 byte mode;
 byte slaveAddress;
 byte slaveRegister;
 byte data;
 unsigned int delayTime;
 byte pin ;// used for tone
 int frequency ;
 int duration ;

 switch (command) {
     case STEPPER_DATA: // Process the command
         ....
```

To support a new device, we need to specify a new SysEx command, to be used by both the Firmata client and server. We also may need to specify a new pin mode.

Adding the New SysEx Command and Pin Mode to FirmataPlus.h

Using your favorite text editor, open **FirmataPlus.h** in the **Arduino sketchbook**. On my system this file is located at:

```
~/Arduino/libraries/FirmataPlus
```

Locate the **User Defined Commands** section of the file. Hexadecimal value 0x72 has not yet been assigned, so we will assign this value to our new command. To be consistent with Firmata naming conventions, the name of the new SysEx command will be **STEPPER_DATA**.

Enter the new command in the file as shown below:

```
/* user defined commands */

#define ENCODER_CONFIG  0x20 // create and enable encoder object
#define ENCODER_DATA            0x21 // current encoder data
#define TONE_DATA               0x5F // request a tone be played
#define SONAR_CONFIG            0x60 // configure a sonar distance
                                    // sensor for operation
#define SONAR_DATA              0x61 // Data returned from sonar
                                    // distance sensor

/* New Command for stepper motor
#define STEPPER_DATA            0x72 // stepper motor command
*/
```

We also need to add a new pin mode for the stepper, and adjust the pin count in this file. There are 13 pin modes, including INPUT and OUTPUT.

```
// pin modes

//#define INPUT         0x00 // defined in wiring.h
//#define OUTPUT        0x01 // defined in wiring.h
#define ANALOG          0x02 // analog pin in analogInput mode
#define PWM             0x03 // digital pin in PWM output mode
#define SERVO           0x04 // digital pin in Servo output mode
#define SHIFT           0x05 // shiftIn/shiftOut mode
#define I2C             0x06 // pin included in I2C setup
#define ONEWIRE         0x07

// New Pin Mode
#define STEPPER          0x08 // Any pin in Stepper mode
//
```

```
#define TONE             0x09 // Any pin in TONE mode
#define ENCODER          0x0a
#define SONAR            0x0b // Any pin in Ping mode
#define IGNORE           0x7f

#define TOTAL_PIN_MODES         13
```

FirmataPlus.h editing is now complete. Save and close the file

Adding the SysEx command to pymata_command_handler.py

Now let's add the new SysEx command (0x72) to PyMata.

Using your favorite editor, open **pymata_command_handler.py** located in the **PyMata-master/PyMata** directory or in the directory where you unzipped the PyMata files.

Locate the Firmata sysex commands section and add the new command.

```
# Firmata sysex commands

SERVO_CONFIG = 0x70  # set servo pin and max and min angles
STRING_DATA = 0x71  # a string message with 14-bits per char

### New command
STEPPER_DATA = 0x72  # Stepper motor command
###


I2C_REQUEST = 0x76  # send an I2C read/write request
I2C_REPLY = 0x77    # a reply to an I2C read request
I2C_CONFIG = 0x78   # config I2C settings such as delay times
REPORT_FIRMWARE = 0x79  # report name and version of the firmware
SAMPLING_INTERVAL = 0x7A  # modify the sampling interval
```

Save and close this file.

Adding the Pin Mode to pymata.py

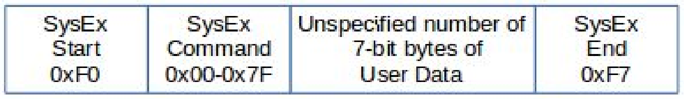
Finally, we need to add the pin mode to pymata.py.

Using your favorite editor, open pymata.py and add the new pin mode.

```
# pin modes

INPUT = 0x00  # pin set as input
OUTPUT = 0x01  # pin set as output
ANALOG = 0x02  # analog pin in analogInput mode
PWM = 0x03  # digital pin in PWM output mode
SERVO = 0x04  # digital pin in Servo output mode
I2C = 0x06  # pin included in I2C setup
ONEWIRE = 0x07  # possible future feature

### New Pin Mode
STEPPER = 0x08  # any pin in stepper mode
###


TONE = 0x09  # Any pin in TONE mode
ENCODER = 0x0a
SONAR = 0x0b  # Any pin in SONAR mode
IGNORE = 0x7f
```

PyMata does not use TOTAL_PIN_MODES, and there it is not added to this file. Save and close this file.

NOTE: Currently only input pin mode types are actually being used by the PyMata code, but we are adding both input and output pin mode pin types to be consistent with the FirmataPlus code.



SysEx Start 0xF0	SysEx Command 0x00-0x7F	Unspecified number of 7-bit bytes of User Data	SysEx End 0xF7

Step 4: Defining the SysEx Protocol Messages

In the previous step, we could have assigned separate SysEx commands for each of the operations that need to be performed to configure and control the motor. However, we chose to assign only one value, STEPPER_DATA, to help conserve the limited number of available SysEx commands.

So, in order to have a STEPPER_DATA command perform one of the three operations specified by our previously defined PyMata API methods, a qualifier byte or subcommand, will be added to the STEPPER_DATA message. The diagram above shows the qualifier bytes that differentiate the STEPPER_DATA commands.

Below, we repeat the API method signatures as a reference, as well as show the protocol specification for each subcommand, byte by byte.

Note that the command protocol messages are originated from PyMata and are sent to FirmataPlus.

Also note that the API method **stepper_request_library_version(),** does not involve the Firmata protocol. It simply returns a previously stored value that is retained by PyMata.

STEPPER_CONFIGURE

API Method Definition:

def stepper_config(self, steps_per_revolution, stepper_pins)

```
// stepper motor configuration message definition for 2 conductor motor

0  START_SYSEX                     (0xF0)
1  STEPPER_DATA                    (0x72)
2  STEPPER_CONFIGURE               (0x00)
3  steps per revolution LSB
4  steps per revolution MSB
5  motor control pin 1
6  motor control pin 2
7  END_SYSEX (0xF7)

// stepper motor configuration message definition for 4 conductor motor

0  START_SYSEX                     (0xF0)
1  STEPPER_DATA                    (0x72)
2  STEPPER_CONFIGURE               (0x00)
3  steps per revolution LSB
4  steps per revolution MSB
5  motor control pin 1
6  motor control pin 2
7  motor control pin 3
8  motor control pin 4
9  END_SYSEX (0xF7)
```

STEPPER_STEP

API Method Definition:

def stepper_step(self, motor_speed, number_of_steps)

```
// stepper motor motion message

0  START_SYSEX                     (0xF0)
1  STEPPER_DATA                    (0x72)
2  STEPPER_STEP                    (0x01)
3  motor speed LSB                 motor speed has a maximum of 21 bits
4  motor speed bits 8-14
5  motor speed MSB
6  number of steps to move LSB
7  number of steps to move MSB
8  motor direction                 determined within stepper_step method
9  END_SYSEX (0xF7)
```

STEPPER_LIBRARY_VERSION

API Method Definition:

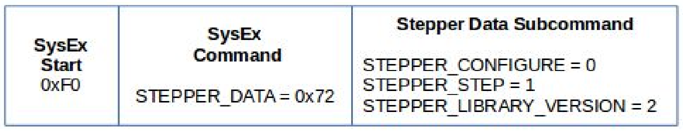
def stepper_request_library_version(self)

```
// stepper motor request library version

0  START_SYSEX                     (0xF0)
1  STEPPER_DATA                    (0x72)
2  STEPPER_LIBRARY_VERSION         (0x02)
3  END_SYSEX                       (0xF7)
```

The reply to the library version request is the only report message sent from the Arduino for the stepper motor. Therefore we can simply use the STEPPER_DATA message without any subcommand qualifier. If there were additional report types for the stepper motor, then we would need to add additional qualifiers for the reports.

Note that the report message is originated from FirmataPlus and sent to PyMata

```
// stepper motor version reply

0  START_SYSEX                     (0xF0)
1  STEPPER_DATA                    (0x72)
2  version LSB
3  version MSB
4  END_SYSEX                       (0xF7)
```



SysEx Start	**SysEx Command**	**Stepper Data Subcommand**

0xF0	STEPPER_DATA = 0x72	STEPPER_CONFIGURE = 0
		STEPPER_STEP = 1
		STEPPER_LIBRARY_VERSION = 2

Step 5: Adding STEPPER_DATA Subcommands to Both the Client and Server Code

Using your favorite text editor, open pymata.py and add the stepper motor subcommands below the tone command section:

```
#  Tone commands
   TONE_TONE = 0  # play a tone
   TONE_NO_TONE = 1  # turn off tone

# Stepper Motor Sub-commands
   STEPPER_CONFIGURE = 0  # configure a stepper motor for operation
   STEPPER_STEP = 1  # command a motor to move at the provided speed
   STEPPER_LIBRARY_VERSION = 2  # used to get stepper library version number
```

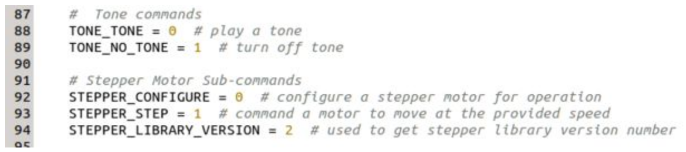
Save and close the file

Now using your text editor, open the FirmataPlus Sketch, FirmataPlus.ino and add the stepper motor subcommands below the Tone commands.

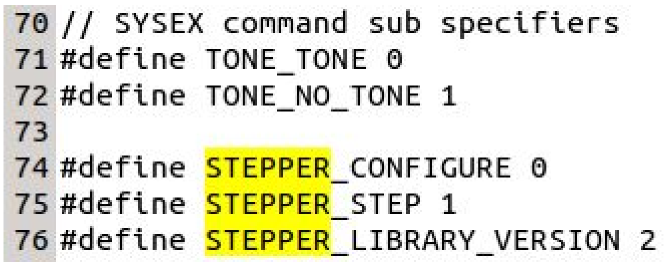
```
// SYSEX command sub specifiers

#define TONE_TONE 0
#define TONE_NO_TONE 1

#define STEPPER_CONFIGURE 0
#define STEPPER_STEP 1
#define STEPPER_LIBRARY_VERSION 2
```

Save and close the file.

Notice that the same values are used by both the client and server.



```
    #  Tone commands
    TONE_TONE = 0  # play a tone
    TONE_NO_TONE = 1  # turn off tone

    # Stepper Motor Sub-commands
    STEPPER_CONFIGURE = 0  # configure a stepper motor for operation
    STEPPER_STEP = 1  # command a motor to move at the provided speed
    STEPPER_LIBRARY_VERSION = 2  # used to get stepper library version number
```



```
// SYSEX command sub specifiers
#define TONE_TONE 0
#define TONE_NO_TONE 1

#define STEPPER_CONFIGURE 0
#define STEPPER_STEP 1
#define STEPPER_LIBRARY_VERSION 2
```

Step 6: Implementing The PyMata API Command and Report Methods

COMMAND MESSAGES - pymata.py

Next we code the PyMata API methods for the three stepper motor sub-commands and an API method to retrieve the reported version number. All code changes for the API are made to pymata.py. The first three methods, stepper_config, stepper_step, and stepper_request_library_version translate the command into a properly formatted SysEx command that is then sent to the Arduino. The last method, get_stepper_version, does not result in any SysEx messaging, but simply returns the result for a previous version request.

```
 def stepper_config(self, steps_per_revolution, stepper_pins):
     """
     Configure stepper motor prior to operation.
     @param steps_per_revolution: number of steps per motor revolution
     @param stepper_pins: a list of control pin numbers - either 4 or 2
     """
     data = [self.STEPPER_CONFIGURE, steps_per_revolution &0x7f, steps_per_revolution >> 7]
     for pin in range(len(stepper_pins)):
         data.append(stepper_pins[pin])
     self._command_handler.send_sysex(self._command_handler.STEPPER_DATA, data)

   def stepper_step(self, motor_speed, number_of_steps):
       """
       Move a stepper motor for the number of steps at the specified speed
       @param motor_speed: 21 bits of data to set motor speed
       @param number_of_steps: 14 bits for number of steps & direction
                               positive is forward, negative is reverse
       """
       if number_of_steps > 0:
           direction = 1
       else:
           direction = 0
       abs_number_of_steps = abs(number_of_steps)
       data = [self.STEPPER_STEP, motor_speed &0x7f, (motor_speed >> 7) &0x7f, motor_speed >> 14,
               abs_number_of_steps &0x7f, abs_number_of_steps >> 7, direction]
       self._command_handler.send_sysex(self._command_handler.STEPPER_DATA, data)

   def stepper_request_library_version(self):
       """
       Request the stepper library version from the Arduino.
       To retrieve the version after this command is called, call
       get_stepper_version
       """
       data = [self.STEPPER_LIBRARY_VERSION]
       self._command_handler.send_sysex(self._command_handler.STEPPER_DATA, data)
```

Finally we implement get_stepper_library_version:

```
def get_stepper_version(self, timeout=20):
       """
       @param timeout: specify a time to allow arduino to process and return a version
```

```
        @return: the stepper version number if it was set.
        """
        # get current time
        start_time = time.time()

        # wait for version to come from the Arduino

        while self._command_handler.stepper_library_version <= 0:
            if time.time() - start_time > timeout:
                print "Stepper Library Version Request timed-out. Did you send a stepper_request_library_version command?"
                return
            else:
                pass
        return self._command_handler.stepper_library_version
```

This completes all the changes to pymata.py. Close and save pymata.py.

REPORT MESSAGES - pymata_command_handler.py

We will now add the report handler method to pymata_command_handler.py, so that it may receive and process stepper library version reports. Notice that this method reassembles the data from the the SysEx message and stores it in an internal variable called stepper_library_version.

```
    def stepper_version_response(self, data):
        """
        This method handles a stepper library version message sent from the Arduino
        @param: data - two 7 bit bytes that contain the library version number
        """
        self.stepper_library_version = (data[0] &0x7f) + (data[1] << 7)
```

Finally, we need to update the command_dispatch table to process the receipt of the stepper version response. Add a new entry to the bottom of the existing table for STEPPER_DATA as shown below. Each entry in the command_dispatch table consists of the SysEx command, the name of the handling method, and the number of 7 bit values returned. The 7 bit values will be reassembled and interpreted as specified by the SysEx message formats we defined earlier.

```
    def run(self):
        """
        This method starts the thread that continuously runs to receive and interpret
        messages coming from Firmata. This must be the last method in this file
        It also checks the deque for messages to be sent to Firmata.
        """
        # To add a command to the command dispatch table, append here.
        self.command_dispatch.update({self.REPORT_VERSION: [self.report_version, 2]})
        self.command_dispatch.update({self.REPORT_FIRMWARE: [self.report_firmware, 1]})
        self.command_dispatch.update({self.ANALOG_MESSAGE: [self.analog_message, 2]})
        self.command_dispatch.update({self.DIGITAL_MESSAGE: [self.digital_message, 2]})
        self.command_dispatch.update({self.ENCODER_DATA: [self.encoder_data, 3]})
        self.command_dispatch.update({self.SONAR_DATA: [self.sonar_data, 3]})
        self.command_dispatch.update({self.STRING_DATA: [self._string_data, 2]})
        self.command_dispatch.update({self.I2C_REPLY: [self.i2c_reply, 2]})
        self.command_dispatch.update({self.CAPABILITY_RESPONSE: [self.capability_response, 2]})
        self.command_dispatch.update({self.PIN_STATE_RESPONSE: [self.pin_state_response, 2]})
        self.command_dispatch.update({self.ANALOG_MAPPING_RESPONSE: [self.analog_mapping_response, 2]})
        self.command_dispatch.update({self.STEPPER_DATA: [self.stepper_version_response, 2]})
```

The command dispatch table is defined as a map in pymata_command_handler.py. These code comments explain its use and how to append new commands to it.

```
# This is a map that allows the look up of command handler methods using a command as the key.

# This is populated in the run method after the python interpreter "sees" all of the
 command handler method defines (python does not have forward referencing)

# The "key" is the command, and the value contains is a list containing the  method name
 and the number of
# parameter bytes that the method will require to process the message (in some cases the
 value is unused)


command_dispatch = {}
```

Save and close pymata_command_handler.py. The client changes are now complete.

Editing:

pymata.py

pymata_command_handler.py

Step 7: Implementing the FirmataPlus Sketch Code

Again, using your text editor, open FirmataPlus.ino and make the following changes:

1. Add stepper.h to the list of "header includes" near the top of the file:

```
#include <Stepper.h>
```

2. Create a pointer to an instance of a stepper motor and set it to NULL. The stepper variable is global, so place it in the Global Variables section of the file.

```
/*==============================================================
       GLOBAL VARIABLES
 ==============================================================*/
// Stepper Motor
Stepper *stepper = NULL;
```

3. Add a case statement to **switch(mode)** in **void setPinModeCallback(byte pin, int mode)** for the stepper pin:

```
case STEPPER:
     pinConfig[pin] = STEPPER ;
     break ;
default:
     Firmata.sendString("Unknown pin mode");
```

4. Add a new case to void sysexCallback() to handle the STEPPER_DATA command and its subcommands.

```
case STEPPER_DATA:
     // determine if this a STEPPER_CONFIGURE command or STEPPER_OPERATE command
     if (argv[0] == STEPPER_CONFIGURE)
     {
       int numSteps = argv[1] + (argv[2] << 7);
       int pin1 = argv[3] ;
       int pin2 = argv[4] ;
       if ( argc == 5 )
       {
         // two pin motor
         stepper = new Stepper(numSteps, pin1, pin2) ;
       }
       else if (argc == 7 ) // 4 wire motor
       {
         int pin3 = argv[5] ;
         int pin4 = argv[6] ;
         stepper =  new Stepper(numSteps, pin1, pin2, pin3, pin4) ;
       }
       else
       {
         Firmata.sendString("STEPPER CONFIG Error: Wrong Number of arguments");
         printData("argc = ", argc) ;
       }
     }
     else if ( argv[0] == STEPPER_STEP )
     {
       long speed = (long)argv[1] | ((long)argv[2] << 7) | ((long)argv[3] << 14);
       int numSteps = argv[4] + (argv[5] << 7);
       int direction = argv[6] ;
       if (stepper != NULL )
       {
         stepper->setSpeed(speed) ;
         if (direction == 0 )
         {
           numSteps *= -1 ;
         }
         stepper->step(numSteps) ;
       }
       else
       {
         Firmata.sendString("STEPPER OPERATE Error: MOTOR NOT CONFIGURED");
       }
     }
     else if ( argv[0] == STEPPER_LIBRARY_VERSION )
     {
       if ( stepper != NULL )
       {
         int version = stepper->version() ;
         Firmata.write(START_SYSEX);
         Firmata.write(STEPPER_DATA);
         Firmata.write(version &0x7F);
         Firmata.write(version >> 7);
         Firmata.write(END_SYSEX);
       }
       else
       {
         // did not find a configured stepper
         Firmata.sendString("STEPPER FIRMWARE VERSION Error: NO MOTORS CONFIGURED");
       }
       break ;
     }
     else
     {
       Firmata.sendString("STEPPER CONFIG Error: UNKNOWN STEPPER COMMAND");
     }
     break ;
```

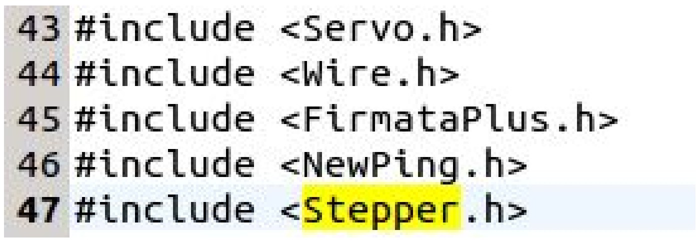
5. In void SystemResetCallback() add the following code for stepper:

```
 // clear stepper pointer
 stepper = NULL ;
```

Close and save the file.

That concludes all of the code changes.

Editing: FirmataPlus.ino



```
#include <Servo.h>
#include <Wire.h>
#include <FirmataPlus.h>
#include <NewPing.h>
#include <Stepper.h>
```

Step 8: Testing the New Functionality

First, using the Arduino IDE, compile and load the modified FirmataPlus.ino Sketch by selecting File/Examples/FirmataPlus that we just modified.

Next we need to install the new version of PyMata we created.

In an administrative Command Window, go to the directory where you extracted PyMata, and type :

```
python setup.py install
```

or for Linux:

```
sudo python setup.py install
```

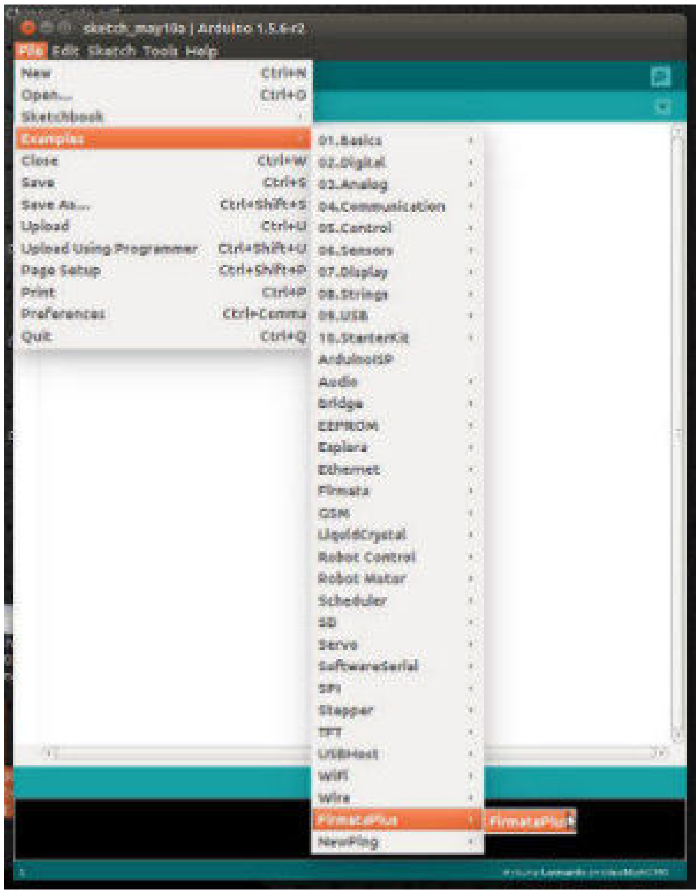
Next we want to run a test script Included with the PyMata distribution. In the PyMata examples directory, there is a python script called pymata_stepper_test.py (the code is shown below). To run this script, go to the examples directory and type:

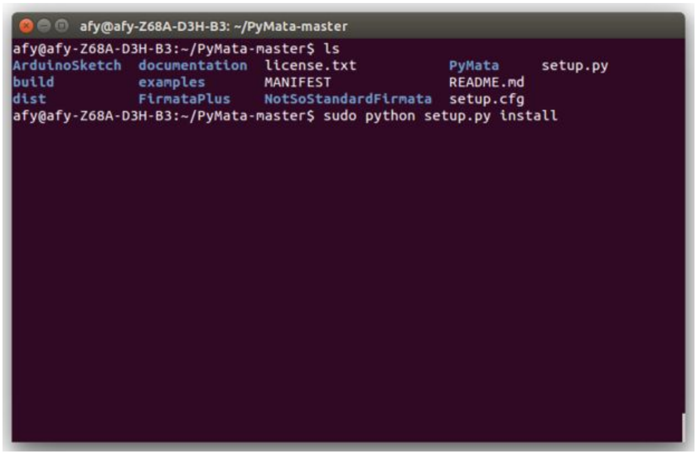
```
python pymata_stepper_test.py
```

As shown in the YouTube video, you should see the motor spin one way and then the reverse.

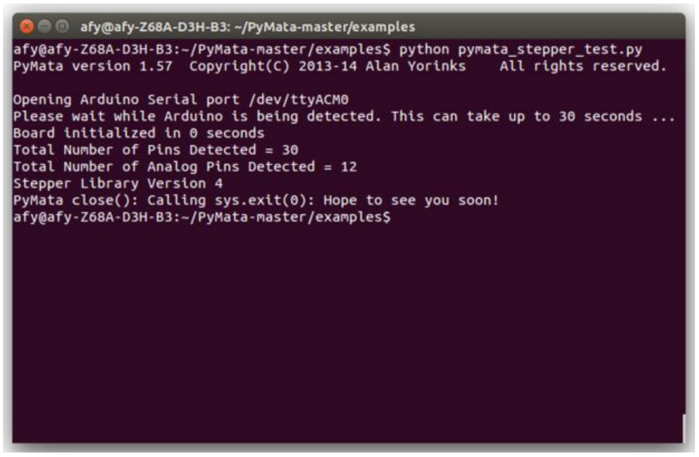
SOURCE for pymata_stepper_test.py

```
from PyMata.pymata import PyMata import time

# Create an instance of PyMata.
firmata = PyMata("/dev/ttyACM0")

# send the arduino a firmata reset
firmata.reset()

# configure the stepper to use pins 9.10,11,12 and specify 512 steps per revolution
firmata.stepper_config( 512, [12, 11, 10, 9])

# allow time for config to complete
time.sleep(.5)

# ask Arduino to return the stepper library version number to PyMata
firmata.stepper_request_library_version()

# allow time for command and reply to go across the serial link
time.sleep(.5)

print "Stepper Library Version",
print firmata.get_stepper_version()

# move motor #0 500 steps forward at a speed of 20
firmata.stepper_step(20, 500)

# move motor #0 500 steps reverse at a speed of 20
firmata.stepper_step(20, -500)

# close firmata
firmata.close()
```

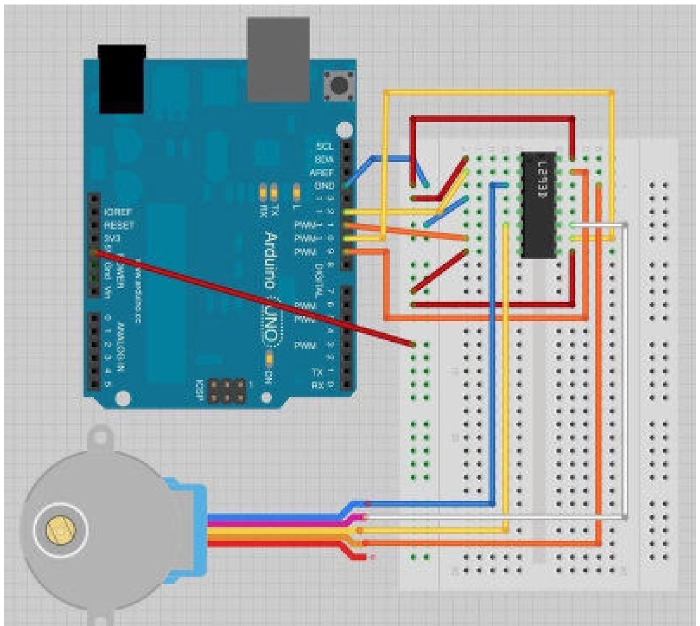




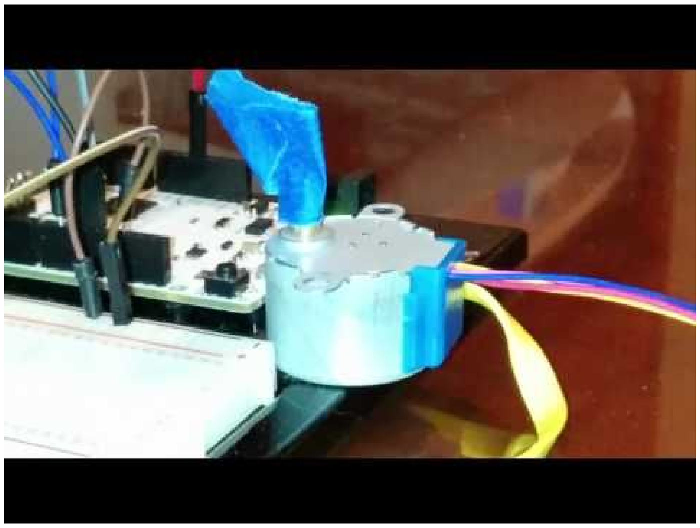
```
afy@afy-Z68A-D3H-B3: ~/PyMata-master
afy@afy-Z68A-D3H-B3:~/PyMata-master$ ls
ArduinoSketch  documentation  license.txt           PyMata     setup.py
build          examples       MANIFEST              README.md
dist           FirmataPlus    NotSoStandardFirmata  setup.cfg
afy@afy-Z68A-D3H-B3:~/PyMata-master$ sudo python setup.py install
```



```
afy@afy-Z68A-D3H-B3: ~/PyMata-master/examples
afy@afy-Z68A-D3H-B3:~/PyMata-master/examples$ python pymata_stepper_test.py
PyMata version 1.57  Copyright(C) 2013-14 Alan Yorinks    All rights reserved.

Opening Arduino Serial port /dev/ttyACM0
Please wait while Arduino is being detected. This can take up to 30 seconds ...
Board initialized in 0 seconds
Total Number of Pins Detected = 30
Total Number of Analog Pins Detected = 12
Stepper Library Version 4
PyMata close(): Calling sys.exit(0): Hope to see you soon!
afy@afy-Z68A-D3H-B3:~/PyMata-master/examples$
```



Drawing Soure: https://learn.adafruit.com/adafruit-arduino-lesson-16-stepper-motors/breadboard-layout



Step 9: Debugging the Arduino Firmata Sketch - Using sendString and printData

Because Firmata utilizes the Ardiuno serial interface, the Arduino IDE serial monitor cannot be used at the same time Firmata is running. This makes debugging a Firmata Sketch difficult to do.

Both StandrardFirmata and FirmataPlus implement the sendString method. This method packages string data and sends it to the client as part of STRING_DATA SysEx message . When PyMata receives a STRING_DATA SysEx message, it prints the message's contents to the Python console.

Sometimes though, we would like to print the current value of an internal Sketch variable accompanied by an identifier of some sort. FirmataPlus to the rescue! The printData function accepts an ID string and a data value as its input parameters, and sends each as STRING_DATA messages to the client. PyMata will print the debug information to the Python console making debugging a little easier.

Here is the FirmataPlus code for printData:

```
void printData(char * id,  long data)
{
 char myArray[64] ;

 String myString = String(data);
 myString.toCharArray(myArray, 64) ;
 Firmata.sendString(id) ;
 Firmata.sendString(myArray);
}
```

Step 10: Now It's Your Turn

We have come to the end of this tutorial, so let's summarize the steps needed to extend Firmata:

1. Spend the time to understand the operation of your selected device. Analyze any associated libraries for the device.

2. Using the knowledge gained in the previous step, craft the new PyMata API method interfaces,

3. Find an available Firmata SysEx command value as well as an available Firmata pin mode value within the Firmata server code. Add these values to both the client and server code.

4. Using step 2 above, identify the subcommands that will be needed to support the new API methods. Do this for both Command and Reporter messages. Add these values to both the client and sever code.

5. Define the new SysEx protocol messages in detail.

6. Implement both client and server code to support the new messages.

7. Test.

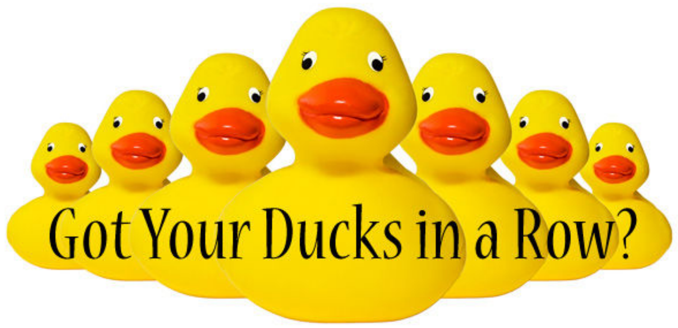
I hope you found this tutorial informative and that you will try extending Firmata on your own.

If you have any questions about this article, or PyMata/FirmataPlus, you may contact me at:

`MisterYsLab@gmail.com`

Thanks for reading!

Alan Yorinks

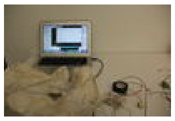


Related Instructables

The Snap!Mobile - Start Your Physical Computing Engines! by MrYsLab

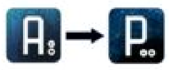
Control a RepStrap with Processing by marc.cryan

Javascript robotics and browser-based Arduino control by danasf



Spacebrew + Vibration Sensors + Arduino + Openframeworks by Snax_and_Macs

Forget Me Not- Remote Flower Watering by Snax_and_Macs



Arduino to Processing: Serial Communication without Firmata by danm_daniel

Comments