Jan 2016 V 0.1.0 beta. Doug Johnson (finson@whidbey.com)

Firmata Feature: DeviceFeature and Device Drivers

Proposed for addition in Firmata 2.6 or later.

The purpose of this feature is to facilitate arbitrary additions to Firmata capabilities without requiring a central registration or causing frequent command code conflicts. The feature is implemented with a new FirmataFeature module `DeviceFeature`, a pair of new Sysex commands (`DEVICE_QUERY` and `DEVICE_RESPONSE`), and the concept of a DeviceDriver abstract class with well defined method signatures.

In effect, `DeviceFeature` uses Firmata as a remote procedure call mechanism.

The `DeviceFeature` module receives, decodes, and dispatches incoming `DEVICE_QUERY` messages to the appropriate device driver. The concrete sub-classes of DeviceDriver implement the various capabilities and make them available to callers through the API documented below. After a request has been processed by the device driver, the `DeviceFeature` module captures the result, encodes, and sends the outgoing `DEVICE_RESPONSE` messages back to the host.

Also note that any other module on the server can use the device driver API directly to access any device capabilities it might require. In this case, there is no reformatting, encoding, transmission, or other involvement by Firmata, it's just one module calling another directly.

Introduction

Device Drivers are designed to allow a client-side application to control remote devices attached to a Firmata server. This is similar to the function of the existing Firmata command set, but at a somewhat higher level of abstraction.

Terminology

Some terms with specific meanings for this feature are *device*, *logical unit* and *handle*.

- Device. A device driver can control one or more instances of a specific *device* type. The capabilities of the driver are determined by the device type, and are equally available to all the instances of the device. Each driver is given a simple name to identify the device type. For example, MCP9808 is the name of a device driver for the MCP9808 temperature sensor. Hello is the name of a virtual device driver that deals with "Hello World" type messaging.
- Logical Unit. Each instance of a device is a *logical unit*. A device driver can handle one or more logical units, depending on device capabilities and device driver implementation. Each logical unit is given a name based on the device name. For example, if there are two MCP9808 boards connected to a single server, then they are referred to as MCP9808:0 and MCP9808:1.
- Handle. When a logical unit is first opened, a *handle* that uniquely identifies the device and logical unit of interest is returned to the caller for use in future operations. The handle is valid until the logical unit is closed.

Device Driver API

The Device Driver API includes six methods documented below. The API is intended to be implemented by a device driver module on the server side (Firmata micro) exactly as written. On the client side (client host), the same API calls should be implemented, but there will be small changes dictated by the syntax of the language used for the client. Client-side proxy device drivers and server-side device drivers always use this API and never compose Firmata messages themselves, instead they rely on Firmata to do that.

In the most common architecture, the device driver implements the main device control code on the server and provides access using the specified API. A proxy on the client also implements the API signatures, and acts as a bridge to the actual device driver and uses the Device Driver Sysex messages DEVICE_QUERY and DEVICE_RESPONSE to control the server side device driver, which in turn controls the component(s) using local capabilities. In this scenario, the server side device driver receives the same calls and parameters as were provided to the proxy on the client.

On the other hand, it is also possible for a device driver to implement the main control code on the client and provide access there using the same API. In this case the client device driver uses existing Firmata Features and commands as necessary to control the remote component(s) directly and according to the data sheet. In this scenario, the server side Firmata responds to standard Firmata commands as received and there is no specific device driver needed on the server.

Device Status and Control Registers

The status and control methods operate based on register numbers. On an actual device, physical register numbers usually start at 0 and max out at a relatively low value like 16 or 32, depending on the device. This DeviceDriver API uses a 16-bit signed integer to identify the register of interest, so virtual quantities and actions can be implemented in addition to the actual physical device capabilities.

Status Return from Methods

Each of the device driver methods returns an `int` value to the caller. If the value is negative, then the call failed and the value is an error code. If the value is greater than or equal to 0, then the call succeeded. The meaning of the non-negative value depends on the call. For example, the `open` method returns a handle for future use and the `read` method returns the number of bytes read. The details for each method are documented below.

Firmata Messages

Two Sysex sub-commands are used by this feature: `DEVICE_QUERY` and `DEVICE_RESPONSE`.

There is a small set of action codes that specify what the driver is to do after it receives the message.

The first action is always `OPEN`. The caller supplies a logical unit name that can be recognized by a device driver, and upon success, a handle is returned for use in future calls. After the handle has been received, the caller can read status (`STATUS`), write control (`CONTROL`), read data stream (`READ`), and write data stream (`WRITE`). Once the caller has completed its operations on a device, it can use `CLOSE` to make the logical unit available for another client.

The detailed message formats for each action are provided at the end of this document.

Method Prototypes

The method prototypes shown below are the primary interface to each Device Driver on the server and, with suitable modifications for language syntax, on the client.

The type identifier `int` is used to indicate a signed integer value of at least 16 bits. Only the low order 16 bits (the two low order bytes) are transmitted for these values by Firmata. The type identifier `byte` is used to indicate an integer value of at least 8 bits. Only the low order 8 bits (one byte) are transmitted for these values by Firmata.

To the extent practical, the error code values and meanings are taken directly from the Linux/C error codes documented in errno.h and errno-base.h, except that the actual values are negated for use in this application.

There are a few parameters whose values are constrained to 14-bit or 7-bit limits because of the way they are transmitted by Firmata. However, they are always presented to the caller as fully sign-extended 16-bit or 8-bit values as documented below. The actual data values being read/written are never constrained because they are always encoded for transmission.

Open

```
int open(const char* name)
int open(const char* name, int flags)
```

param (in) `name` Name of the logical unit to open. UTF-8 encoded, null terminated.

param (in) `flags` Flags associated with the open. Default: 0.

return *Success*: The newly assigned handle value. The handle is used in future calls to indicate the device driver and specific device being addressed. *Error*: error code.

Status

Read information from a register (or virtual register) in the device or device driver.

The method and its parameters are as follows.

```
int status(int handle, int reg, int count, byte *buf)
```

param (in) `handle` The device driver selector value returned by Open in a previous call.

param (in) `reg` The register address at which to start reading.

param (in) `count` The number of bytes to read.

param (out) `buf` Pointer to the buffer to receive the data read. Must be large enough to hold `count` bytes.

return *Success*: The number of bytes actually read. A short count does not in itself cause an error, since the caller can determine that not everything requested was read which may not actually be an error. *Error*: error code.

Control

```
int control(int handle, int reg, int count, byte *buf)
```

param (in) `handle` The device driver selector value returned by Open in a previous call.

param (in) `reg` The register address at which to start writing.

param (in) `count` The number of bytes to write.

param (in) `buf` Pointer to the buffer containing the data to write.

return *Success*: The number of bytes actually written. Ordinarily, this will be equal to the requested number of bytes to write. If it is short due to some device error (eg, physical write failure), then the driver will return an error code (eg, `EIO`). However, under some unique circumstances for some drivers, it may be reasonable for a short count to occur in which case the driver will return the short count and no error code. *Error*: error code.

Read

```
int read(int handle, int count, byte *buf)
```

param (in) `handle` The device driver selector value returned by Open in a previous call.

param (in) `count` The number of bytes to read.

param (out) `buf` Pointer to the buffer to receive the data read. Must be large enough to hold `count` bytes.

return *Success*: The number of bytes actually read. A short count does not in itself cause an error, since the caller can determine that not everything requested was read which may not actually be an error. *Error*: error code.

Write

```
int write(int handle, int count, byte *buf)
```

param (in) `handle` The device driver selector value returned by Open in a previous call.

param (in) `count` The number of bytes to write.

param (in) `buf` Pointer to the buffer containing the data to write. Must contain at least `count` bytes.

return *Success*: The number of bytes actually written. Ordinarily, this will be equal to the requested number of bytes to write. If it is short due to some device error (eg, physical write failure), then the driver will return an error code (eg, `EIO`). However, under some unique circumstances for some drivers, it may be reasonable for a short count to occur in which case the driver will return the short count and no error code. *Error*: error code.

Close

```
int close(int handle)
```

param (in) `handle` The device driver selector value returned by Open in a previous call. The selected device driver is responsible for deciding what actions if any are needed to "close" the connection. After a close, the only valid action on the device is another open.

return Success: 0. *Error*: error code.

Message Formats

The arguments provided by the caller of an API method are formatted into a `DEVICE_QUERY` message on the client side by the proxy device driver, then transmitted to the server. Firmata dispatches the Sysex message to the DeviceFeature module, which decodes it and dispatches the API call to the proper device driver. After processing by the device driver, DeviceFeature captures the results and formats them as a `DEVICE_RESPONSE` message, and transmits the message back to the client host where the proxy device driver decodes the message and returns the result to the original caller.

In the case of header values, the high order bit in a byte must be 0 since these values are transmitted without any encoding. In the case of parameter blocks (the actual values that are read/written from/to the devices) there is no restriction on the values because these values are encoded in Base-64 before transmission.

Header

The `DEVICE_QUERY` and `DEVICE_RESPONSE` message headers are Sysex message bytes 0 to 7.

DEVICE_QUERY header

```
0 START_SYSEX byte (0xF0).
1 Sysex command byte DEVICE_QUERY (0x30).
2 Device Action byte, with values as described below.
3 Reserved (0)
4 LSB of the 14-bit flags or handle value.  The highest order bit is 0.
5 MSB of the 14-bit flags or handle value.  The highest order bit is 0.
6 Reserved (0)
7 Reserved (0)
```

DEVICE_RESPONSE header

```
0 START_SYSEX byte (0xF0).
1 Sysex command byte DEVICE_RESPONSE (0x31).
2 Device Action byte that was provided in the associated DEVICE_QUERY.
3 Reserved (0)
4 LSB of the 14-bit handle value.  The highest order bit is 0.
5 MSB of the 14-bit handle value.  The highest order bit is 0.
6 LSB of the 14-bit return/status value.  The highest order bit is 0.
7 MSB of the 14-bit return/status value.  The highest order bit is 0.
```

Device Action Types

These are 7-bit values, stored in Firmata `DEVICE_QUERY` and `DEVICE_RESPONSE` messages at offset 2.

```
OPEN    (0x00)
STATUS  (0x01)
CONTROL (0x02)
READ    (0x03)
WRITE   (0x04)
CLOSE   (0x05)
```

Flags or Handle

These are signed 14-bit values, stored in the Firmata `DEVICE_QUERY` and `DEVICE_QUERY` messages at offsets 4 and 5. The values are stored on the client in a single, wider integer variable (`int16_t`, `int32_t`, etc).

```
4 flags (LSB, bit 7 = 0)
5 flags (MSB, bit 7 = 0)
```

or

```
4 handle (LSB, bit 7 = 0)
5 handle (MSB, bit 7 = 0, bit 6 = 0 (sign bit))
```

Status / Return Value from Methods

Each of the device driver methods returns an `int` value to the caller. The meaning of the returned `int` varies depending on the method called. Handles, byte counts, and error status returns are all handled by Firmata the same way. Handle values and byte counts are always positive, to distinguish them from error return values.

For transmission by Firmata, the `int` being returned is considered to be a 14-bit signed integer. The low-order 7 bits are put in the LSB, and bit 7 is set to 0. The higher-order 6 bits and the sign bit are put in the MSB, and bit 7 is set to 0. The resulting two bytes are stored in the header at offsets 6 and 7. The value is reassembled and sign extended by Firmata on the client side before passing it back to the original caller.

Note that the byte count returned by the various methods is the number of actual bytes read or written, it is *not* the length of the encoded message body. Once the message body is decoded back to the raw values on the client, the two lengths will again be equal. The encode/decode should all happen outside the view of the caller, so this won't be a problem except as something to remember when debugging and looking at the messages as they are transmitted.

Parameter Block

The parameter block contains the extra information needed to complete a request such as register numbers, byte counts, and the actual data read or written.

The parameter block is transmitted in the body of the message (all bytes after offset 7 except the final END_SYSEX). This block is encoded before transmission using an 8-bit to 7-bit encoder. The standard encoder is Base-64. This encode/decode is handled entirely by the Firmata libraries right before and after transmission of the Sysex messages and should not ordinarily be visible to the client application.

Character strings are stored on the server in UTF-8. All eight bits in a UTF-8 byte are significant. A '0' in the high order bit indicates a character in the first group of 127 characters (the ASCII character set). A '1' in the high order bit indicates that the byte is part of a multi-byte sequence. Unfortunately, it might also indicate a Firmata control byte. Encoding in Base-64 avoids this problem.

In the following message tables, the message contents are all shown one byte per row. Remember that all bytes starting at offset 8 are encoded prior to transmission. The values shown in the tables below starting at offset 8 are **the 8-bit values before or after encoding / decoding**, they are ***not*** the 7-bit quantities that are actually transmitted.

Detailed Device Driver Message Formats

Device Driver - Open

Query

> **Message Header** (Plain text)
>
> ```
> 0  START_SYSEX (0xF0)
> 1  DEVICE_QUERY (0x30)
> 2  0x00 (OPEN)
> 3  0 (Reserved)
> 4  flags (LSB)
> 5  flags (MSB)
> 6  0 (Reserved)
> 7  0 (Reserved)
> ```
>
> **Parameter Block** (before encoding)
>
> > ```
> > 0..n name string (UTF-8)
> > ```
>
> **Message End** (Plain text)

```
k  END_SYSEX (0XF7)
```

Response

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_RESPONSE (0x31)
2  0x00 (OPEN)
3  0 (Reserved)
4  0 (Reserved)
5  0 (Reserved)
6  return/status (LSB)
7  return/status (MSB)
```

Message End (Plain text)

```
8  END_SYSEX (0XF7)
```

Device Driver - Status

Query

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_QUERY (0x30)
2  0x01 (STATUS)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  0 (Reserved)
7  0 (Reserved)
```

Parameter Block (encoded during transmission with Base-64)

```
0  count (LSB)
1  count (MSB)
2  register (LSB)
3  register (MSB)
```

Message End (Plain text)

```
16  END_SYSEX (0XF7)
```

Response

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_RESPONSE (0x31)
2  0x01 (STATUS)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  return/status (LSB)
7  return/status (MSB)
```

Parameter Block (encoded during transmission with Base-64)

```
0..n Status data bytes read, if any
```

Message End (Plain text)

```
k  END_SYSEX (0XF7)
```

Device Driver - Control

Query

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_QUERY (0x30)
2  0x02 (CONTROL)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  0 (Reserved)
7  0 (Reserved)
```

Parameter Block (encoded during transmission with Base-64)

```
0  count (LSB)
1  count (MSB)
2  register (LSB)
3  register (MSB)
4..n control bytes to write
```

Message End (Plain text)

```
k  END_SYSEX (0XF7)
```

Response

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_RESPONSE (0x31)
2  0x02 (CONTROL)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  return/status (LSB)
7  return/status (MSB)
```

Message End (Plain text)

```
8  END_SYSEX (0XF7)
```

Device Driver - Read

Query

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_QUERY (0x30)
2  0x03 (READ)
```

```
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  0 (Reserved)
7  0 (Reserved)
```

Parameter Block (encoded during transmission with Base-64)

```
0  count (LSB)
1  count (MSB)
```

Message End (Plain text)

```
12  END_SYSEX (0XF7)
```

Response

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_RESPONSE (0x31)
2  0x03 (READ)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  return/status (LSB)
7  return/status (MSB)
```

Parameter Block (encoded during transmission with Base-64)

```
0..n Data bytes read, if any
```

Message End (Plain text)

```
k  END_SYSEX (0XF7)
```

Device Driver - Write

Query

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_QUERY (0x30)
2  0x04 (WRITE)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  0 (Reserved)
7  0 (Reserved)
```

Parameter Block (encoded during transmission with Base-64)

```
0  count (LSB)
1  count (MSB)
2..n data bytes to write
```

Message End (Plain text)

```
k  END_SYSEX (0XF7)
```

Response

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_RESPONSE (0x31)
2  0x04 (WRITE)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  return/status (LSB)
7  return/status (MSB)
```

Message End (Plain text)

```
8  END_SYSEX (0XF7)
```

Device Driver - Close

Query

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_QUERY (0x30)
2  CLOSE (0x05)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  0 (Reserved)
7  0 (Reserved)
```

Message End (Plain text)

```
8  END_SYSEX (0XF7)
```

Response

Message Header (Plain text)

```
0  START_SYSEX (0xF0)
1  DEVICE_RESPONSE (0x31)
2  CLOSE (0x05)
3  0 (Reserved)
4  handle (LSB)
5  handle (MSB)
6  return/status (LSB)
7  return/status (MSB)
```

Message End (Plain text)

```
8  END_SYSEX (0XF7)
```