Proposals

From Firmata

This is a place to propose extensions to the Firmata protocol

Contents

PinModes

```
#define ONEWIRE                  0x07 // pin configured for 1-wire
#define STEPPER                  0x08 // pin configured for stepper motor
#define IGNORE                   0x7F // pin configured to be ignored by digitalWrite and capabilityRespons
#define TOTAL_PIN_MODES          10
```

SysEx Protocol Extensions

The following sysex commands are proposed for future versions of Firmata. For a list of currently implemented sysex commands please see the Protocol (http://firmata.org/wiki/Protocol) page.

```
#define ERROR_CODE          0x68 // send error code to client
#define STEPPER_DATA       0x72 // configure, step and speed commands
#define ONEWIRE_DATA       0x73 // OneWire read/write/reset/select/skip/search request and reply to read/se
#define PULSE_DATA         0x74 // pulseIn/pulseOut data message (34 bits)
#define SHIFT_DATA         0x75 // shiftIn/shiftOut data message
#define SCHEDULER_DATA     0x7B // send createtask/deletetask/addtotask/schedule/querytasks/querytask reque
                                // receive querytasks/querytask-request from the scheduler
#define CONFIG_EXT         0x7C // further configuration not covered by pin_mode
```

Error Msg Proposal

Error messages are currently sent as strings (see StandardFirmata). These strings use a lot of SRAM. It would be better to create a new sysex message for sending error codes. A client library developer would then provide a lookup table to the error codes, storing the strings in their client code. The error codes messages would be documented in the Firmata source as well as on Firmata.org.

```
 * -----------------------------
 * 0  START_SYSEX (0xF0)
 * 1  ERROR_CODE   (0x68)
 * 2  errorCode   (0-127)
 * 3  END_SYSEX   (0xF7)
```

Then define a constant for each error:

```
#define UNKNOWN_PIN_MODE       0  // "Unknown pin mode."
#define I2C_TOO_MANY_BYTES     1  // "I2C Read Error: Too many bytes received."
#define I2C_TOO_FEW_BYTES      2  // "I2C Read Error: Too few bytes received."
#define I2C_10BIT_ADDR         3  // "I2C Error: 10-bit addressing mode not supported."
#define I2C_TOO_MANY_QUERIES   4  // "I2C Error: Too many queries."
```

Could alternatively use constant byte values or enum.

See corresponding issue in github: https://github.com/firmata/arduino/issues/78

Hardcoded Text Query

Query a constant Sring, hardcoded into the sketch. Is intended for usage as 'metadata' store for software to query. E.g. A home automation software would like to know what relays can be grouped together as "Room 1" or "Room 2", or what set of output belongs to a "smart kettle" or a "smart microwave" and for each object, map the "buttons" to each GPIO. (As for why. It is so that the experience is more like plug and play for some softwares like Home Automation Softwares, while still giving flexibility to just trigger I/O as a standard firmata)

I don't know if this is how to make a query, please modify to make it suit.

```
// This is how to represent a string constant array in C. This will be empty in a new firmata source.
const hardCodedString = "OSA relay bank V1.0 , kettle:ON=GPIO1 , kettle:OFF=GPIO2";
```

```
/* Query Firmware Name and Version
 * 0  START_SYSEX (0xF0)
 * 1  queryHardCodedStrings (0x79)
 * 2  END_SYSEX (0xF7)
 */
```

my Comment on this: I'd rather see a facility to store and retrieve persistent data in/from eeprom without modifying the source. Maybe key/value pairs (whereas the key might just be an 14bit int and value plain binary data not restricted to strings). This would be beneficial in other use-cases (e.g. IP-configuration for EthernetFirmata) as well. - NTruchsess

I also think eeprom is a better way to store a data. There was discussion in the past of using eeprom for storing the state of the board. You'll see some comments regarding this in StandardFirmata. I think the intention was to restore the board state in the event of an intentional reset during a long running process. I think this is a use case that stemmed more from the use of firmata in art installations where you would have an exhibition running uninterrupted for a month or more. However I also really like this use case of eeprom for general purpose storage. Not sure if there is enough eeprom to handle both cases or not, but I'm leaning in favor of general purpose storage. - Jeff Hoefs 2/19/13

pulseIn/pulseOut Proposal

```
/* pulseIn/Out (uses 32-bit value)
 * -------------------------------
 * 0  START_SYSEX (0xF0) (MIDI System Exclusive)
 * 1  pulseIn/Out (0x74)
 * 2  dataPin (0-127)
 * 3  bits 0-6 (least significant byte)
 * 4  bits 7-13
 * 5  bits 14-20
 * 6  bits 21-27
 * 7  bits 28-34 (most significant byte)
 * 8  END_SYSEX (0xF7) (MIDI End of SysEx - EOX)
 */
```

ShiftIn/Out Proposal

Supporting shift registers should be possible without much difficulty now that sysex has been implemented.

```
/* shiftIn/Out (uses 8-bit value)
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  shiftOut (0x75)
 * 2  dataPin (0-127)
 * 3  clockPin (0-127)
 * 4  latchPin (0-127)
 * 5  msbFirst (boolean)
 * 6  bits 0-6 (least significant byte)
 * 7  bit 7 (most significant bit)
 * n  ... (as many byte pairs as needed)
 * n+1  END_SYSEX (0xF7)
 */
```

Or perhaps this should be more like the servo messages, with a `SET_PIN_MODE`-specific mode, like `SHIFT`, then a `SHIFT_CONFIG` message. An issue with this approach is if you have multiple shift registers you will need to store the data, clock and latch pins for each register in a structure. You would have to allocate memory to store these structures in an array.

```
/* shiftIn/Out config
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  SHIFT_CONFIG
 * 2  dataPin (0-127)
 * 3  clockPin (0-127)
 * 4  latchPin (0-127)
 * 5  msbFirst (boolean)
 * 6  END_SYSEX (0xF7)
 */
```

```
/* shiftIn/Out data (total bits chopped to multiple of 8 so extra bits are ignored)
 * ------------------------------
 * 0   START_SYSEX (0xF0)
 * 1   SHIFT_DATA
 * 2   dataPin (0-127)
 * 3   bits 0-6 (least significant byte)
 * 4   bits 7-13
 * 5   bits 14-20
 * 6   bits 21-27
 * n   ...
 * n+1 END_SYSEX (0xF7)
 */
```

Here's another approach (updated 6/6/13). It's similar to the first proposal but adds a numBytes parameter to shift in and also adds a shift reply for sending requested shift in data back to the client application. This version leaves out the latch pin. The advantage of leaving it out is that it would be easier to shift in or out an arbitrary number of bytes. The disadvantage is that the latchPin toggle need to be sent in 2 separate data packets (standard digital out messages). There is an example of this proposal implemented here: https://github.com/firmata/arduino/pull/51.

```
// shift out
* 0  START_SYSEX
* 1  SHIFT_DATA (0x75)
* 2  SHIFT_OUT (0x01)
* 3  dataPin
* 4  clockPin
* 5  bitOrder (MSBFIRST or LSBFIRST)
* n ... (shift out data)
* n+1 END_SYSEX

// shift in (for client application to request shift-in data from microcontroller)
* 0  START_SYSEX
* 1  SHIFT_DATA (0x75)
* 2  SHIFT_IN (0x02)
* 3  dataPin
* 4  clockPin
* 5  bitOrder (MSBFIRST or LSBFIRST)
* 6  numBytes (number of bytes to shift in. Default to 1)
* 7 END_SYSEX

// shift in reply (for sending shift-in data to client application)
* 0  START_SYSEX
* 1  SHIFT_DATA (0x75)
* 2  SHIFT_IN_REPLY (0x03)
* 3  dataPin  (so you know which data pin the reply corresponds to)
* n ... (shift in data)
* n+1 END_SYSEX
```

Stepper Motor Proposal

Listed below is the current implementation as created for the Maxuino project. It is useable only for stepper drivers that use the STEP and DIRECTION input configuration (most of the mainstream ones fall into this catagory). It will not work for those rolling their own directly controlled steppers (using darlington arrays and such). Feedback appreciated.

```
/* Stepper CONFIG
 * --------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Stepper Command (0x67)
 * 2  config command (0)    //0=config subcommand, 1=step subcommand, 2=speed subcommand
 * 3  device# (0-5)  //supporting 6 motors at the moment
 * 4  steps-per-revolution lsb (least significant byte)
 * 5  steps-per-revolution msb  (most significant bit)  //the common rage is 100-400 but I am unsure of ho
 * 6  directionPin# (0-127)
 * 7  stepPin# (0-127)
 * 8  END_SYSEX (0xF7)
 */
```

```
/* Stepper STEP
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Stepper Command (0x67)
 * 2  config command (1)
 * 3  device# (0-5)
 * 4  #steps lsb (least significant byte)
 * 5  #steps msb  (most significant byte)  //it is not uncommon to need to send tens of thousands of steps
 * 6  direction (0-1)   // 0 = negative, 1 = positive
 * 7  END_SYSEX (0xF7)
 */
```

```
/* Stepper SPEED
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Stepper Command (0x67)
 * 2  config command (2)
 * 3  device# (0-5)
 * 4  #speed lsb (least significant byte) // speed in revs per minute
 * 5  #speed msb  (most significant byte)
 * 6  END_SYSEX (0xF7)
 */
```

Alternate approach. This version supports both step + dir stepper drivers (EasyDriver, etc.) as well as 2 and 4 wire drivers (H-bridge, darlington array, etc). Jeff Hoefs 10/27/12. This is also the version of the stepper protocol currently implemented in the configurable (https://github.com/firmata/arduino/blob/configurable/utility/StepperFirmata.cpp) branch of Firmata.

```
/* Stepper CONFIG
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Stepper Command (0x72)
 * 2  config command (0)    //0=config subcommand, 1=step subcommand, 2=speed subcommand
 * 3  device# (0-5)  //supporting 6 motors at the moment
 * 4  interface  // 1 = Step + Dir driver, 2 = two wire configuration, 4 = four wire configuration
 * 5  steps-per-revolution lsb (least significant byte)
 * 6  steps-per-revolution msb  (most significant bit)  //the common range is 100-400 but I am unsure of
 * 7  motorPin1 or directionPin# (0-127)
 * 8  motorPin2 or stepPin# (0-127)
 * 9  motorPin3 (0-127) [only when interface = 4]
 * 10 motorPin4 (0-127) [only when interface = 4]
 * 11 END_SYSEX (0xF7)
 */
```

```
/* Stepper STEP
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Stepper Command (0x72)
 * 2  config command (1)
 * 3  device# (0-5)
 * 4  direction (0-1)   // 0 = CW, 1 = CCW
 * 5  #steps byte1 // LSB
 * 6  #steps byte2
 * 7  #steps byte3 // MSB  21 bits (2,097,151 steps max)
 * 8  speed lsb  // steps in 0.01*rad/sec  (2050 = 20.50 rad/sec)
 * 9  speed msb
 * 10 accel lsb [optional]  // acceleration in 0.01*rad/sec^2 (1000 = 10.0 rad/sec^2)
 * 11 accel msb [optional]
 * 12 decel lsb [optional]  // deceleration in 0.01*rad/sec^2
 * 13 decel msb [optional]
 * 14 END_SYSEX (0xF7)
 */
```

Query Pull-up Resistor Status Proposal

When a digital pin is in `INPUT` mode, setting the pin `HIGH` will turn on the internal pull-up resistors. It could be useful to query the state of that pin somehow.

Please note: This is already possible using the pin state query (http://firmata.org/wiki/Protocol#Pin_State_Query). If an internal pull-up is set for a pin, the pin state query will return 1 otherwise it will return 0.

set digital pin mode INPUT_PULLUP Pull-up

Use a new digital pin mode INPUT_PULLUP with a value of (TBD) to activate the internal pull-up resistor. Benefit (IMHO), it shows directly, without interpreting other values (see Query Pull-up Resistor Status Proposal) that the pull-up is activated on that pin. Arduino uses the INPUT_PULLUP constant for PinMode(), too. "CaptBlaubaer"

```
/* set digital pin mode
 * --------------------
 * 1  set digital pin mode (0xF4) (MIDI Undefined)
 * 2  pin number (0-127)
 * 3  state (INPUT/INPUT_PULLUP/OUTPUT/ANALOG/PWM/SERVO, 0/tbd/1/2/3/4)
 */
```

SPI Proposal

There has been some preliminary work done on including SPI in Firmata. It would most likely be based on SysEx.

Generic Scheduler Proposal

The idea is to store a stream of messages on arduino which is replayed later (either once or repeated). A task is created by sending a create_task message. The time-to-run is initialized with 0 (which means the task is not yet ready to run). After filling up the taskdata with messages (using add_to_task command messages) a final schedule_task request is send, that sets the time-to-run (in milliseconds after 'now'). If a task itself contains delay_task or schedule_task-messages these cause the execution of the task to pause and resume after the amount of time given in such message has elapsed. If the last message in a taks is a delay_task message the task is scheduled for reexecution after the amount of time specified. If there's no delay_task message at the end of the task (so the time-to-run is not updated during the run) the task gets deleted after execution.

```
/* Scheduler CREATE_TASK request
 * -----------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  create_task command (0x00)
 * 3  taskid (0-127)
 * 4  length LSB (bit 0-6)
 * 5  length MSB (bit 7-13)
 * 6  END_SYSEX (0xF7)
 */
```

```
/* Scheduler DELETE_TASK request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  delete_task command (0x01)
 * 3  taskid (0-127)
 * 4  END_SYSEX (0xF7)
 */
```

```
/* Scheduler ADD_TO_TASK request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  add_to_task command (0x02)
 * 3  taskid (0-127)
 * 4  taskdata bit 0-6   [optional] //task bytes encoded using 8 times 7 bit for 7 bytes of 8 bit
 * 5  taskdata bit 7-13  [optional]
 * 6  taskdata bit 14-20 [optional]
 * n  ... // as many bytes as needed (don't exceed MAX_DATA_BYTES though)
 * n+1  END_SYSEX (0xF7)
 */
```

```
/* Scheduler DELAY_TASK request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  delay_task command (0x03)
 * 3  time_ms bit 0-6 //time_ms is of type long, requires 32 bit.
 * 4  time_ms bit 7-13
 * 5  time_ms bit 14-20
 * 6  time_ms bit 21-27
 * 7  time_ms bit 28-31
 * 8  END_SYSEX (0xF7)
 */
```

```
/* Scheduler SCHEDULE_TASK request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  schedule_task command (0x04)
 * 3  taskid (0-127)
 * 4  time_ms bit 0-6 //time_ms is of type long, requires 32 bit.
 * 5  time_ms bit 7-13
 * 6  time_ms bit 14-20
 * 7  time_ms bit 21-27
 * 8  time_ms bit 28-31
 * 9  END_SYSEX (0xF7)
 */
```

```
/* Scheduler QUERY_ALL_TASKS request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  query_all_tasks command (0x05)
 * 3  END_SYSEX (0xF7)
 */
```

```
/* Scheduler QUERY_TASK request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  query_task command (0x06)
 * 3  taskid (0-127)
 * 4  END_SYSEX (0xF7)
 */
```

```
/* Scheduler RESET request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  scheduler reset command (0x07)
 * 3  END_SYSEX (0xF7)
 */
```

```
/* Scheduler ERROR_FIRMATA_TASK reply
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  error_task Reply Command (0x08)
 * 3  taskid (0-127)
 * 4  time_ms bit 0-6
 * 5  time_ms bit 7-13
 * 6  time_ms bit 14-20
 * 7  time_ms bit 21-27
 * 8  time_ms bit 28-31 | (length bit 0-2) << 4
 * 9  length bit 3-9
 * 10 length bit 10-15 | (position bit 0) << 7
 * 11 position bit 1-7
 * 12 position bit 8-14
 * 13 position bit 15 | taskdata bit 0-5 << 1 [taskdata is optional]
 * 14 taskdata bit 6-12  [optional]
 * 15 taskdata bit 13-19 [optional]
 * n  ... // as many bytes as needed (don't exceed MAX_DATA_BYTES though)
 * n+1  END_SYSEX (0xF7)
 */
```

```
/* Scheduler QUERY_ALL_TASKS reply
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  query_all_tasks Reply Command (0x09)
 * 3  taskid_1 (0-127) [optional]
 * 4  taskid_2 (0-127) [optional]
 * n  ... // as many bytes as needed (don't exceed MAX_DATA_BYTES though)
 * n+1  END_SYSEX (0xF7)
 */
```

```
/* Scheduler QUERY_TASK reply
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  Scheduler Command (0x7B)
 * 2  query_task Reply Commandc (0x0A)
 * 3  taskid (0-127)
 * 4  time_ms bit 0-6
 * 5  time_ms bit 7-13
 * 6  time_ms bit 14-20
 * 7  time_ms bit 21-27
 * 8  time_ms bit 28-31 | (length bit 0-2) << 4
 * 9  length bit 3-9
 * 10 length bit 10-15 | (position bit 0) << 7
 * 11 position bit 1-7
 * 12 position bit 8-14
 * 13 position bit 15 | taskdata bit 0-5 << 1 [taskdata is optional]
 * 14 taskdata bit 6-12  [optional]
 * 15 taskdata bit 13-19 [optional]
 * n  ... // as many bytes as needed (don't exceed MAX_DATA_BYTES though)
 * n+1  END_SYSEX (0xF7)
 */
```

OneWire Proposal

The idea is to configure Arduino Pins as OneWire Busmaster. The may be more than one pin configured for

OneWire and there may be more than one device connected to such a pin. Each one-wire-device has a unique identifier which is 8 bytes long and comes factory-programmed into the the device. To scan all devices connected to a pin configured for onewire a SEARCH-request message is sent. The response contains all addresses of devices found. Having the address of a device OneWire-command-messages may be sent to this device. The actual commands executed on the OneWire-bus are 'reset', 'skip', 'select', 'read', 'delay' and 'write' All these commands may be executed with a single OneWire-command-message. The subcommand-byte contains these commands bit-encoded. The data required to execute each bus-command must only be included in the message when the corresponding bit is set. The order of execution of bus commands is: 'reset'->'skip'->'select'->'write'->'read'->'delay' (remember: each of these steps is optional. Also some combinations don't make sense and in fact are mutual exclusive in terms of OneWire bus protocol, so you cannot run a 'skip' followed by a 'select') The delay is useful for OneWire-commands included into taskdata (see Firmata-scheduler proposal). Some OneWire-devices require some time to carry out e.g. a a/d-conversion after receiving the appropriate command. Including a delay into a OneWire-message saves some bytes in the taskdata (in comparism to the inclusion of a 'delay_task' scheduler message). OneWire Read- and ReadReply messages are correlated using a correlationid (16bits). The reply contains the correlationid-value that was sent with the original request.

```
/* OneWire SEARCH request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  OneWire Command (0x73)
 * 2  search command (0x40|0x44) //0x40 normal search for all devices on the bus. 0x44 SEARCH_ALARMS reque
 * 3  pin (0-127)
 * 4  END_SYSEX (0xF7)
 */
```

```
/* OneWire SEARCH reply
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  OneWire Command (0x73)
 * 2  search reply command (0x42|0x45) //0x42 normal search reply. 0x45 reply to a SEARCH_ALARMS request
 * 3  pin (0-127)
 * 4  bit 0-6   [optional] //address bytes encoded using 8 times 7 bit for 7 bytes of 8 bit
 * 5  bit 7-13  [optional] //1.address[0] = byte[0]    + byte[1]<<7 & 0x7F
 * 6  bit 14-20 [optional] //1.address[1] = byte[1]>>1 + byte[2]<<6 & 0x7F
 * 7  ....                 //...
 * 11 bit 49-55                    //1.address[6] = byte[6]>>6 + byte[7]<<1 & 0x7F
 * 12 bit 56-63            //1.address[7] = byte[8]    + byte[9]<<7 & 0x7F
 * 13 bit 64-69            //2.address[0] = byte[9]>>1 + byte[10]<<6 &0x7F
 * n  ... // as many bytes as needed (don't exceed MAX_DATA_BYTES though)
 * n+1  END_SYSEX (0xF7)
 */
```

```
/* OneWire CONFIG request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  OneWire Command (0x73)
 * 2  config command (0x41)
 * 3  pin (0-127)
 * 4  power (0-1) // 1 = leave pin on state high after write to support parasitic power, 1 = don't leave 
 * 5  END_SYSEX (0xF7)
 */
```

```
/* OneWire COMMAND request
 * ------------------------------
 * 0  START_SYSEX (0xF0)
 * 1  OneWire Command (0x73)
 * 2  command bits (0x00-0x2F) // bit 0 = reset, bit 1 = skip, bit 2 = select, bit 3 = read, bit 4 = dela
 * 3  pin (0-127)
 * 4  bit 0-6   [optional] //data bytes encoded using 8 times 7 bit for 7 bytes of 8 bit
 * 5  bit 7-13  [optional] //data[0] = byte[0]   + byte[1]<<7 & 0x7F
 * 6  bit 14-20 [optional] //data[1] = byte[1]>1 + byte[2]<<6 & 0x7F
 * 7  ....                 //data[2] = byte = byte[2]>2 + byte[3]<<5 & 0x7F ...
 * n  ... // as many bytes as needed (don't exceed MAX_DATA_BYTES though)
 * n+1  END_SYSEX (0xF7)
 */

/* data bytes within OneWire Request Command message
 * ------------------------------
 *  0 address[0]                    [optional, if bit 2 set]
 *  1 address[1]                              "
 *  2 address[2]                              "
 *  3 address[3]                              "
 *  4 address[4]                              "
 *  5 address[5]                              "
 *  6 address[6]                              "
 *  7 address[7]                              "
 *  8 number of bytes to read (LSB) [optional, if bit 3 set]
 *  9 number of bytes to read (MSB)           "
 * 10 request correlationid byte 0            "
 * 11 request correlationid byte 1            "
 * 10 delay in ms (bits 0-7)        [optional, if bit 4 set]
 * 11 delay in ms (bits 8-15)                 "
 * 12 delay in ms (bits 16-23)                "
 * 13 delay in ms (bits 24-31)                "
 * 14 data to write (bits 0-7)      [optional, if bit 5 set]
 * 15 data to write (bits 8-15)               "
 * 16 data to write (bits 16-23)              "
 * n  ... // as many bytes as needed (don't exceed MAX_DATA_BYTES though)
 */
```

```
/* OneWire READ reply
 * 0  START_SYSEX (0xF0)
 * 1  OneWire Command (0x73)
 * 2  read reply command (0x43)
 * 3  pin (0-127)
 * 4  bit 0-6   [optional] //data bytes encoded using 8 times 7 bit for 7 bytes of 8 bit
 * 5  bit 7-13  [optional] //correlationid[0] = byte[0]   + byte[1]<<7 & 0x7F
 * 6  bit 14-20 [optional] //correlationid[1] = byte[1]>1 + byte[2]<<6 & 0x7F
 * 7  bit 21-27 [optional] //data[0] = byte[2]>2 + byte[3]<<5 & 0x7F
 * 8  ....                 //data[1] = byte[3]>3 + byte[4]<<4 & 0x7F
 * n  ... // as many bytes as needed (don't exceed MAX_DATA_BYTES though)
 * n+1  END_SYSEX (0xF7)
 */
```

Extended Configuration Proposal

for special configuration not covered in pinmode-specific messages

```
/* set analogReference request
 * 0  START_SYSEX (0xF0)
 * 1  CONFIG_EXT command (0x7C)
 * 2  ANALOG_REF command (0x00)
 * 3  analog reference source (0-4, 0=DEFAULT, 1=INTERNAL, 2=INTERNAL1V1, 3=INTERNAL2V56, 4=EXTERNAL
 *    Values supported are device-specific.
 *    See http://arduino.cc/en/Reference/AnalogReference and Arduino.h for details)
 */
```

```
/* set analogReadResolution request
 * 0  START_SYSEX (0xF0)
 * 1  CONFIG_EXT      command (0x7C)
 * 2  ANALOG_READ_RES command (0x01)
 * 3  analog read resolution (1-32, see http://arduino.cc/en/Reference/AnalogReadResolution for details)
 */
```

```
/* set analogWriteResolution request
 * 0  START_SYSEX (0xF0)
 * 1  CONFIG_EXT       command (0x7C)
 * 2  ANALOG_WRITE_RES command (0x02)
 * 3  analog write resolution (1-32, see http://arduino.cc/en/Reference/AnalogWriteResolution for details
 */
```

comment: We should definitely have something like this in place before support for Arduino Due is added. I've added an issue for this to github: https://github.com/firmata/arduino/issues/39

Digital write pin alongside of digital message port

With the current digital message port, the host needs to reset or query pin state, then keep state in sync.

However with new implementations, like my BLE implementation, I dont want to force a reset on connect as BLE connects and disconnects possibly often. Instead Id have to do ~29 pin state queries to build an internal data structure, but thats hugely wasteful on an RF connection!

It also seems odd, anyway, to keep state which could get out of sync, when the board is already keeping state-- It can read its register settings any time!

I understand why the digitalmessage port was implemented like it was, it is clever to send 8 pins for 3 bytes. But in this scenario it can be equally or more efficient to send 4 bytes for 1 pin if it means we dodge ~29 pin state queries and config writes.

It seems like its time to have a digital message pin command in addition to the digitalMessagePort which could just move a single pin when state is unknown, or when you don't want to hold state in the host program.

We would have use up a sysex command

```
/* set digital pin
 * 0  START_SYSEX (0xF0)
 * 1  digital write pin (0x??)
 * 2  high/low (0x01)
 * 3  END_SYSEX (0xF7) (MIDI End of SysEx - EOX)
 */
```

```
Another option actually, is to put optional on digital message port
```

```
/* two byte digital data format, second nibble of byte 0 gives the port number (e.g. 0x92 is the third po
 * 0  digital data, 0x90-0x9F, (MIDI NoteOn, but different data format)
 * 1  digital pins 0-6 bitmask
 * 2  digital pin 7 bitmask
 * 3  digital pins 0-6 bitmask pins you want to effect
 * 4  digital pin 7 bitmask    pins you want to effect
 */
```

Building and dependencies

I've searched for Firmata java library not depending from Processing sources to use in JaxaFX 2.0 or in Android projects. So i does not exist at this moment(I suggest to:

```
1. create Firmata class out of existing Arduino, which will work with ...
2. ISerial interface
3. create adapters (Processing, regular Java, Android) which implement ISerial interface and use concrete
```

Also suggest to use Maven as build tool as it is java wide used practice. Can make it myself (dev [at] antonsmirnov [dot] name)

Retrieved from "http://firmata.org/wiki/Proposals"