Firmata: Towards making microcontrollers act like extensions of the computer

Hans-Christoph Steiner
Interactive Telecommunications Program, New York University
hans@at.or.at

Abstract

Firmata is a generic protocol for communicating with microcontrollers from software on a host computer. The central goal is to make the microcontroller an extension of the programming environment on the host computer in a manner that feels natural in that programming environment. It was designed to be open and flexible so that any programming environment can support it, and simple to implement both on the microcontroller and the host computer to ensure a wide range of implementations. The current reference implementation is a library for Arduino/Wiring and is included with Arduino software package since version 0012. There are matching software modules for a number of languages, like Pd, OpenFrameworks, Max/MSP, and Processing.

Keywords: arduino, microcontroller, pure data, processing, python

1. Introduction

Firmata began in 2006 as a demo for Arduino that I created while at STEIM for a residency. I had followed devices like Eroktronix MIDItron [1] which aimed to be an easily configurable microcontroller for musical uses. I had a number of Arduinos on hand and needed to control Pd with some sensors. After trying a number of different setups, I rapidly tired of constantly reprogramming the Arduino for each configuration.

In my work with microcontrollers, they were always tied to a computer via a serial protocol over a wire. The microcontrollers were used to get data in from sensors and output control data out to motors, relays, etc. When used this way, it is necessary to have a serial protocol for communicating between the microcontroller and the host computer. Since the rest of the projects was developed in a single programming environment, it became apparent that the microcontroller should behave as an extension of that programming environment rather than a distinct unit with its own method of being programmed.

Development began with the Arduino microcontroller board, and that is still the main development environment. The reference Firmata library has also been ported to the whole Arduino family as well as the Wiring board. Any Arduino- or Wiring-compatible board can directly use the existing Firmata firmwares [1] and library. Work is underway to port it to the closely-related Sanguino [2] board.

2. Previous Work

Before Firmata, there were a number of projects also aimed to turn a standardized microcontroller board into an easily configurable sensor input/output box. After the initial prototype, I evaluated a number of these in order to compare the various approaches. In these, I saw examples of how to make things straightforward to use and configure, which then inspired the design of Firmata.

STEIM was an early developer of "sensor box" devices. The Sensorlab was an sensor-voltage-to-MIDI converter, and the Junxion Box carried that idea to the next level using USB HID. The I-CubeX was an early example of a device that was simple to connect sensors to and get the data in the computer, inspired by the STEIM Sensorlab. The I-CubeX extended the possibilities of communication beyond just MIDI.

Phidgets [4] extended this "sensor box" idea by making a range of different hardware configurations for different I/O needs. Each Phidgets board has a fixed number of I/O pins which it reports via the USB HID protocol. This allows the boards to work without installing drivers, and gets the data to the host software using the standard mechanism that operating systems use to get data from the human to the computer: USB HID [8] and associated APIs. The downside of Phidgets is that the boards are not configurable and have their inputs and outputs fixed in hardware.

The MIDItron took this idea to the next step by allowing for configuration via the provided Max/MSP patch. The MIDItron uses MIDI both for the configuration and communication. If used in an existing MIDI setup, it is very straightforward to integrate in with other MIDI equipment. The configuration options make it more flexible than the similar devices.

3. Review of Protocols

[1] a firmware is a program, usually very small, burned into flash memory, like on a microcontroller
[2] http://sanguino.cc/

In the process of developing the Firmata protocol, many existing protocols were reviewed. The core goals in designing the protocol where: simplicity over super-flexibility, efficient bandwidth usage, and ease of implementation. Therefore, the focus was on finding an existing protocol that could be re-purposed.

3.1. ASCII protocols

First off, ASCII-based protocols considered since they are more easily human readable. There are a number of simple, ASCII-based protocols out there which are generally pretty easy to implement and understand. For the sake of compatibility, the Arduino is limited to classic serial bitrates, so the maximum is 115,200 bits/second. One ASCII protocol considered is the *Simple Message System* (SMS) [3] for Arduino. The protocol established for the USB Bit Whacker [2] is also an ASCII protocol similar to SMS. They are relatively easy to read and to implement in text-based programming environments. The first problem is that they tend to require many more bytes. For example, SMS needs 4 bytes to set one digital pin, 56 bytes to set all digital pins. This verbosity also makes it much more difficult to implement full duplex communication and multitasking, since the microcontroller has to spend so much time handling the serial I/O. As a comparison, the current Firmata protocol needs 3 bytes to set one digital pin or all digital pins.

And lastly, ASCII protocols are actually more difficult to implement in Pd or Max/MSP, and perhaps other languages relevant to NIME that do not have strong string handling capabilities. Therefore ASCII protocols were ruled out since they would slow down the processing a lot and limit the use in musical controllers because of the latency and jitter.

The problem is not so apparent with a steady stream, like a continuous stream of analog values. The core problem is when there is a lot of intermittent data, like button presses, in between the steady stream. That chunk of intermittent data could block the stream, adding jitter. If everything is moving fast enough, then the jitter wouldn't be noticeable. If rhythmic button presses are slowed down by a large amount of the analog input data, then performance would degrade. An ASCII-based protocol would also have to be zero-padded. Otherwise, you would have 2 bytes for analog values less than 10 and 5 bytes for analog values ¿=1000, causing a pretty wide range of jitter.

3.2. Open Sound Control

At first thought, Open Sound Control (OSC) [3] seemed like a natural choice for the basis of Firmata. It is widely used and implemented and not too difficult to use. Björn Hartmann already had a working OSC implementation for the Arduino [5], which was consulted during the design phase. The first problem was that OSC was designed with much faster connections in mind, e.g. network connections. The minimum size of an OSC "bundle" is 24 bytes, and that just sends one value. While the packet size for OSC over serial can be a lot smaller, 11 bytes minimum (1 byte header, 1 byte for packet size, data in 4-byte chunks, type tag in 4 bytes, 1-byte checksum), this is still substantially larger than the 3 byte MIDI packets. Also, OSC's larger, more complicated packets require more resources to handle the packets, perhaps not important on the host computer, but definitely a concern on microcontrollers. Lastly, OSC is not easy to implement, especially on microcontrollers. Part of the goal of the Firmata project is to expand to other platforms beyond the Atmel AVR and the Arduino. [4]

3.3. Gainer

The protocol established for the GAINER[6] project was also considered. It is based on call and response somewhat similar to USB HID. The host sends a request, the Gainer board replies. The message size varies from 1 byte to 6 bytes. Only the reply messages from the microcontroller have terminators, the calls from the host computer are just a single byte. While this approach does have the advantage of an easily controllable poll time, the asymmetry of the messages made the protocol more complicated. Perhaps more importantly, since it is a custom protocol, there are not many reference implementations to draw from.

3.4. USB HID

The USB HID protocol is very efficient and widespread, all modern operating systems support it, most programming environments have USB HID APIs, it uses minimal bandwidth, and was designed to be implemented on microcontrollers. But it is very difficult to implement, even understanding the basics is non-trivial. After years of struggling with USB HID and its inordinate complexity and reams of official documentation, [7] it was clear that USB HID is a bloated and baroque protocol specification.

Also, the Arduino can not be a proper USB HID device because it uses USB-Serial, though it would be possible to use the HID packets on top of USB-Serial. I opted not to use it because it is a very complicated and obfuscated protocol itself, and the APIs provided by Microsoft and Apple are as complicated as the USB HID protocol itself. While USB HID would provide high performance digital and analog I/O, using USB HID as a protocol for sending configuration messages like Firmata does would be very tricky to implement properly with USB HID.

3.5. MIDI

MIDI is a relatively easy to implement, efficient, and widespread. The resolution of the analog messages is only 7-bits, which can be quite limiting. The core MIDI messages range from 1 to 3 bytes. SysEx messages can theoretically be any length. The initial Firmata protocol was quite similar to MIDI, so it

[3] http://archive.cnmat.berkeley.edu/OpenSoundControl/OSC-spec.html

[4] There is currently a PIC implemention of an earlier version of the protocol.

made it easy to port Firmata to use a MIDI compatible message format. MIDI has a 7-bit command space, which is also used for channel information as well. Since the core MIDI messages have a limited command space and a 3 byte message limit, Firmata makes use of MIDI SysEx messages for configuration messages, which are sent much less frequently than analog and digital data. For some data types like pulseIn pulse measurement which produce 32-bits of data or more, the data is sent using a specific SysEx message.

3.6. SLIP

One protocol which was not examined as part of the initial design was SLIP [5] . It is simple, efficient, and easy to implement. It is a very minimal protocol, so something like MIDI message types would have to be added on top in order to fulfill the design goals of Firmata. If there is pressing need for a major protocol revision, SLIP could replace MIDI to simplify the Firmata protocol further still. Making Firmata a SLIP based protocol has the potential of making it even simpler to read and implement, but at the cost of having to reimplement everything. In the end that would probably not be worth it.

4. Design

Following these experiences and the experience of using the Arduino with Pd for musical instruments, it became apparent that the next step was to create a standard protocol to represent both the Arduino API and the types of data that would be transferred between the microcontroller and the host computer. Usability was put first and foremost in the design, even at the expense of some performance.

One concern of many Arduino developers was that using MIDI made it difficult for non-technical people to understand the Firmata code. Having a whole firmware, protocol, and host software that a beginner can understand is definitely a worthy and laudable goal. But it did not seem possible to achieve efficiency high enough for use in musical controllers while keeping the protocol and code simple enough for most Arduino users to understand. Instead the goal was more akin to TCP/IP. Beginners use them all the time and they are a robust and efficient. In most environments, they are quite straightforward to use. But the underlying protocols are quite complicated and far from understandable for even an advanced beginner. People use them because programming languages provide good interfaces, not because the protocols themselves are easy to understand.

MIDI was chosen as the core data protocol since it is efficient, relatively easy to implement, and there was a lot of existing implementations freely available for repurposing. Only the MIDI message format is used, not the whole MIDI protocol. Instead of using the standard MIDI message interpretations (NoteOn, Aftertouch, etc), a new set of interpretations was devised to represent both the data types (14-

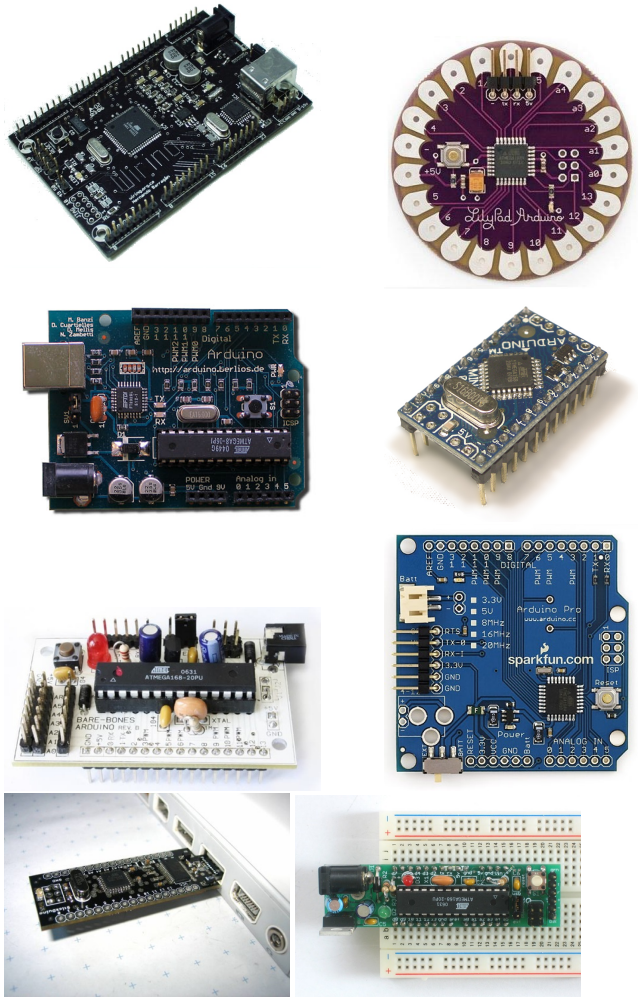


Figure 1. Some devices that work with Firmata (left column: Wiring, Arduino NG, Bare Bones Arduino, Stickduino; and, right column: Arduino Lilypad, Arduino Mini, Arduino Pro, Boarduino.

[5] Serial Line Internet Protocol

bit analog and digital) and the control messages (pinMode, PWM, etc). This makes it possible to represent the Arduino API using Firmata messages. The command mappings here will not be directly usable in terms of MIDI controllers and synths. It should co-exist with MIDI without trouble and can be parsed by standard MIDI interpreters. As it stands now, Firmata can represent 16 analog inputs at 14-bit resolution, and 128 digital pins.

One thing that the MIDI standard did quite well was it provided a set of standard interpretations of data, so that it was possible to plug in and have control over volume, pitch, bend, and even the type of instrument. MIDI does this without requiring that the messages be interpreted in specific ways; this opens it up to a wide range of uses. So while OSC has proven a much more flexible data transmission protocol, the lack of standardized messages and interfaces means that OSC does not address this important aspect of MIDI. With Firmata, a central part of the project is to provide a full API for common operations involving microcontrollers, while leaving things wide open for unstructured exploration. Each pin is set to a model analog input, digital input, digital output, PWM output, servo control, shift register control, and LED matrix controllers are supported. The `pinMode` message and other specific configuration messages are used to switch each pin between these options. This collection of modes is something similar to the standard TCP/IP port numbers: a table of numbers which each represent a specific "service".

4.1. Bitrates

As part of the process of preparing the `Standard_Firmata` firmware to be the default firmware on new Arduino boards, we needed to choose a default bitrate. Since a central aim was to support musical controller design, the bitrate needed to be relatively fast. Initially the bitrate was set at 57600 since some users reported dropped packets at 115200, but through testing we found that the error rates were comparable for 57600 and 115200. Since the Arduino Bluetooth requires 115200 for its virtual serial port, 115200 was chosen as the default bitrate.

The introduction of the Arduino Xbee caused us to revisit this issue, since it wants to use a bitrate of 111111. We compared the Arduino's 16MHz clock rate to serial clock rates and found that 125000 is the ideal bitrate, and 111111 has very low error rates. The limitations of the FTDI USB-serial driver forced the issue, meaning that for general release, only the standard rates could be used. For users who need lower error rates, 38400 is a better option.

There is also a possibility of errors with very high speed, bidirectional traffic in echo tests. At 115200, it's possible to send a byte every 0.069 ms, so there problem lies elsewhere. Our best estimate based on the testing is that this problem is caused by the microcontroller's processor being overloaded with tasks and therefore not able to keep up with the serial data coming in. While these errors are possible to reproduce with very specific test setups, we have found that in real world use, they are rarely a problem. Perhaps more importantly, it is most likely not a problem with the protocol, as MIDI is well proven, therefore there is the possibility for software and hardware improvements as a remedy.

4.2. Life Without Zero

Some programming environments, most notably Adobe Flash [6], can only communicate using null-terminated string types. So while individual characters could easily be interpreted as their binary values, 0 was not available, since it was reserved to mean the end of a string. The binary value of 0 is reserved to represent the end of the string. There was early discussion on how best to support this case in the protocol itself, since Flash is commonly used in conjunction with Arduinos. In the end it was agreed that the serial proxy that was already required for Flash to communicate with the Arduino should just be modified to convert the binary protocol to something that made sense in terms of null-terminated strings.

5. Using Firmata

5.1. Firmata as Library

For the Arduino and Wiring platforms, Firmata is implemented as a library. That means that custom firmwares can use the Firmata protocol so that they can be controlled from the host computer using the existing Firmata software. With the Arduino software, the library is built in and includes a collection of example firmwares to guide users to developing their own custom firmwares.

5.2. Code Examples

Here are three very simple examples of using native library to access a microcontroller running a Firmata-based firmware. These examples all assume that the microcontroller is attached to the first serial port.

5.2.1. Processing

```
import processing.serial.*;
import cc.arduino.*;

Arduino arduino;

void setup() {
  arduino = new Arduino(this, Arduino.list()[0]);
  arduino.pinMode(5, Arduino.INPUT);
}

void draw() {
  if (arduino.digitalRead(5) == Arduino.HIGH)
    println("Digital pin 5 is HIGH");
  else
    println("Digital pin 5 is LOW");
  print("Analog pin 0 value is ");
  println(arduino.analogRead(0));
}
```

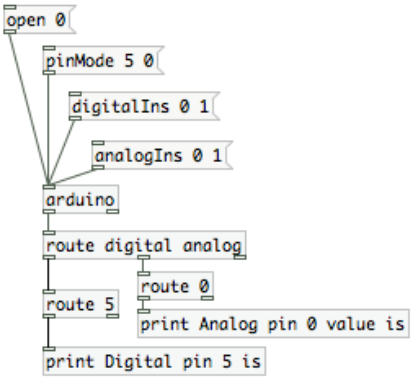
[6] as far as I know, this has changed as of ActionScript 3

5.2.2. Python

```
import pyduino, sys, time

arduino = pyduino.Arduino(0)
arduino.digital[5].set_mode(pyduino.DIGITAL_INPUT)
arduino.digital_ports[0].set_active(1)
arduino.analog[0].set_active(1)
while 1:
   arduino.iterate()
   value = arduino.digital[5].read()
   if value == 1:
       print "Digital pin 5 is HIGH"
   else:
       print "Digital pin 5 is LOW"
   value = arduino.analog[0].read()
   print "Analog pin 0 value is %f"% value
```

5.2.3. Pure Data



5.3. What Works Now

Firmata started out as a single firmware for Arduino. As users of Firmata had more and more ideas for what to do with it, it became clear that it should be an Arduino/Wiring library instead of just a single firmware. The original single firmware then became `Standard_Firmata`. This firmware is meant to include as much functionality as possible into a single firmware. Since it is not always easy nor possible to include new functions into the big `Standard_Firmata`, new firmwares can be created using the Firmata library. `Standard_Firmata` and some example firmwares are included in the Arduino environment. This includes a simple implementation of a firmware that routes analog messages to the instances of the Arduino `Servo` library.

As of this writing, the version of `Standard_Firmata` included in Arduino 0014 includes support for digital input and output, analog input, PWM output, switching pins between digital input and digital output. Additionally, there are messages that control the reporting of analog and digital inputs, so that it is possible to receive only the inputs that are of interest. There are other versions of `Standard_Firmata` which include support for servos, LED matrices, poll-time

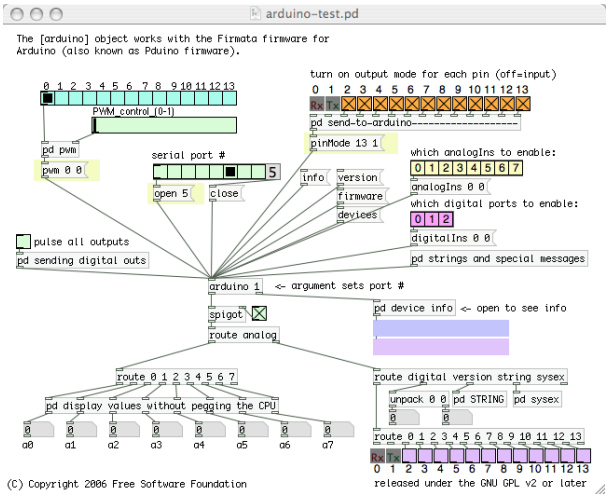


Figure 2. The arduino-test.pd patch which outlines the possibilities with Pd+Firmata

configuration, and I^2C. Over time, as this code proves stable and unintrusive to other functions, it will be incorporated into the reference `Standard_Firmata`.

All of these messages are part of the protocol and the library, but the authors of Firmata implementations on the host computer are encouraged to make that software behave as naturally as possible in that programming environment. For example, the Processing Arduino library uses an event callback interface that is common in that environment while Pd's [arduino] outputs messages on its outlet as it receives them. Not all implementations even expose the controls over the input reporting, instead they automatically send those messages when the user requests data from that input.

5.4. Users in the Real World

Firmata has already seen wide use in the Processing and Pd user communities, and has recently started to see more use with Flash, OpenFrameworks, Max/MSP, and Python. Firmata was also an integral part of the course work for classes such as Björn Hartmann and Bill Verplank's HCI courses at Stanford's Institute of Design [7], Zach Lieberman and Ayah Bdeir's Making Things Move course at Parsons School of Design [8], and the author's NIME class at ITP/NYU [9]. Through the process of incorporating Firmata in their course work, all of them have become contributors to the development and expansion of the protocol, the libraries and the firmwares. This has lead to the development of the `firmata.org` website, a set of online resources to open up the further development.

[7] http://protolab.pbwiki.com/
[8] http://makingthingsmove.org/
[9] http://itp.nyu.edu/nime/

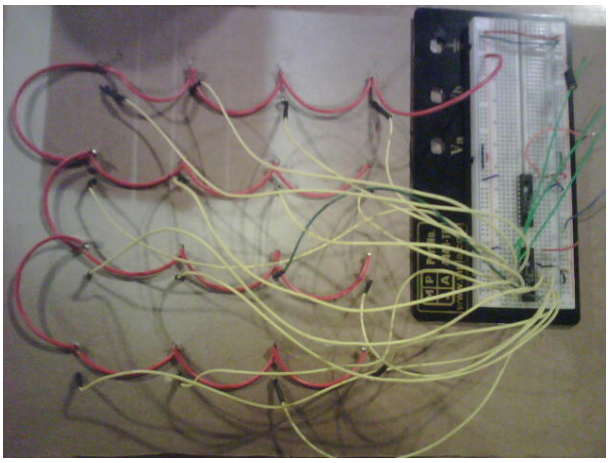


Figure 3. Ayah Bdeir's Firmata shift register setup controlling a matrix of LEDs

6. Future Work

The next milestone for Firmata is make `Standard_Firmata` the default firmware that is installed on every new Arduino board. Additionally, we are working on expanding the flexibility of Firmata by extending the Arduino library to be transport-neutral. That enables seamless use of Firmata over USB-serial, bluetooth, Xbee, and even ethernet. There are working implementations of Firmata over all of those transports, the hard part is making all o them coexist in the same library.

Also, the standard protocol elements for more specific things like PWM have proven quite useful, so the aim is to create a registry where people can add a wide range of specific additions to the protocol, allowing plug-and-play programming of things like SPI and I^2C devices, and other standard pieces of electronics using in physical computing.

Lastly, since the current reference Firmata implementation was completed in February 2008, the micro-OSC firmware was published at NIME 2008. Since I do not use OSC at in my own work, I have not followed its development closely. I am interesting in how the managed the USB connection. The Arduino's USB-serial interface is a source of latency and jitter problems, so the micro-OSC could serve as an example of how to use different USB classes with a microcontroller.

7. Acknowledgements

While I am the sole author of this paper, Firmata would never have gotten far without the contributions of many other people. Jamie Allen helped get the first redesign underway and coined the name "Firmata". Tom Igoe, David Mellis, Massimo Banzi, Shigeru Kobayashi, Erik Sjödin, Björn Hartmann, Casey Reas, and many others provided lots of valuable discussion, feedback on the Arduino Developers' list. Björn Hartmann exposed me to pair programming and we got the first SysEx and Servo support working. NYC Resistor provided a community for me to take this to the next level. Adam Mayer taught me about writing C++ classes. Joe Turner wrote PyDuino; David Mellis wrote the Processing implementation; Erik Sjödin wrote the ActionScript/Flash implementation; Zach Lieberman and Ayah Bdeir wrote the Open Frameworks implementation; Marius Schebella wrote the Max/MSP implementation; Eirik Arthur Blekesaune wrote the SuperCollider implementation; and, "dkapell" wrote the Perl implementation. Last but not least, thanks to the Arduino Team for their support of my work on Firmata. I am sure I forgot some people that I should credit, please accept my apologies in advance.

References

[1] MIDItron. `http://eroktronix.com/`.

[2] Documentation for Firmware D, version 1.4.3 for USB Bit Whacker Boards, 2007. `http://greta.dhs.org/UBW/Doc/FirmwareDDocumentation_v140.html`.

[3] T. O. Fredericks. Simple message system. `http://www.arduino.cc/playground/Code/SimpleMessageSystem`.

[4] S. Greenberg and C. Fitchett. Phidgets: Easy development of physical interfaces through physical widgets. In *Proceedings of the ACM UIST Symposium on User Interface Software and Technology*. ACM Press, 2001.

[5] B. Hartmann. d.tools: Arduino support. `http://hci.stanford.edu/dtools/arduino.html`.

[6] S. Kobayashi, T. Endo, K. Harada, and S. Oishi. Gainer: a reconfigurable i/o module and software libraries for education. In *Proceedings of the 2006 conference on New Interfaces for Musical Expression (NIME'06)*, pages 346–351. IRCAM Centre Pompidou, 2006.

[7] H.-C. Steiner, D. Merrill, and O. Matthes. A unified toolkit for accessing human interface devices in Pure Data and Max/MSP. In *Proceedings of the 2007 conference on New Interfaces for Musical Expression (NIME'07)*. New York University, 2007.

[8] USB Implementers' Forum. Universal Serial Bus (USB) Device Class Definition for HID 1.11, 2001. `http://www.usb.org/developers/devclass_docs/HID1_11.pdf`.