

A Unified Toolkit for Accessing Human Interface Devices in Pure Data and Max/MSP

Hans-Christoph Steiner
IDM/Polytechnic University
Brooklyn, NY, USA
hans@at.or.at

David Merrill
MIT Media Lab
Cambridge, MA, USA
dmerrill@media.mit.edu

Olaf Matthes
nullmedium
Greifswald, Germany
olaf@nullmedium.de

ABSTRACT

In this paper we discuss our progress on the HID toolkit, a collection of software modules for the Pure Data and Max/MSP programming environments that provide unified, user-friendly and cross-platform access to human interface devices (HIDs) such as joysticks, digitizer tablets, and stomp-pads. These HIDs are ubiquitous, inexpensive and capable of sensing a wide range of human gesture, making them appealing interfaces for interactive media control. However, it is difficult to utilize many of these devices for custom-made applications, particularly for novices. The modules we discuss in this paper are `[hidio]`¹, which handles incoming and outgoing data between a patch and a HID, and `[input_noticer]`, which monitors HID plug/unplug events. The goal in creating these modules is to preserve maximal flexibility in accessing the input and output capabilities of HIDs, in a manner that is approachable for both sophisticated and beginning designers. This paper documents our design notes and implementation considerations, current progress, and ideas for future extensions to the HID toolkit.

1. INTRODUCTION AND MOTIVATION

Human Interface Devices (HIDs²) such as joysticks, digitizer tablets, keyboards, mice, gamepads and ‘stomp-pads’ have become widely available and inexpensive. Most existing HIDs are built robustly, and due to their prevalence and low cost, there is growing interest in utilizing them for musical control and other performance applications.

¹a word in square brackets denotes a Max/Pd object

²Human Interface Device (HID) has become the standard term to describe devices designed to sense physical human-computer input, and to send feedback from computers to users. The term ‘HID’ most commonly refers to the USB-HID specification, which is a USB device class that describes human interface devices that utilize USB communication. We use ‘HID’ more generally, to mean the larger set of human interface devices that may or may not use USB.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

NIME07, June 7-9, 2007 New York, NY, USA
Copyright 2007. Copyright remains with the authors.

For an electronic musical instrument designer, easy access to gestural data (motion, pressure, buttonpresses, etc.) and output capabilities (lights, force feedback) enables rapid prototyping of musical affordances and mapping strategies. Many HIDs are temporally and gesturally sensitive enough for musical performance, including gaming mice, certain joysticks, and most graphics tablets. Another factor that makes many existing HIDs appealing for electronic music performances is that they are relatively familiar objects (as compared to custom electronic hardware), which can allow an audience to more easily understand the connection between a performer’s actions and the resulting sonic output. A number of contemporary musicians have used standard HIDs as musical controllers. Leon Gruenbaum’s Samchillian Tip Tip Tip Cheeepeeeee [10] is built upon a standard ergonomic keyboard; Luke Dubois plays the Wacom tablet with The Freight Elevator Quartet [8]; August Black started by hacking existing devices to make his El Lechero [6]; Loic Kessous has built his instrument using a Wacom tablet and a joystick [11].

In addition to the ranks of experienced professionals leveraging HIDs, there is a growing number of students and other novices around the world interested in ‘physical computing’ and its application to media art. Several universities now offer courses specifically in physical computing and/or the design of electronic music controllers [16] [7] [3] [2]. For students, the utilization of HIDs for expressive control of sound, video, or other media can be an appealing alternative to the construction of custom hardware, an endeavor that requires a much greater level of technical sophistication. A number of prototyping platforms have become available lately that also support HID standards, such as CUI [14] and Phidgets [9].

As well as input capabilities, some HIDs provide auditory, visual or haptic feedback. Our goal is to provide easy-to-use input and output access to a wide range of HIDs, through a unified, cross-platform, standardized and coherent approach.

`[hidio]` is a software object for Pure Data and Max/MSP that provides unified and standardized access to HIDs. Pure Data and Max/MSP are popular environments for implementing physical interfaces for musical performance or other media-rich interactive applications. Our goal in creating `[hidio]` is that the end user of these environments will not have to learn how to use a separate object for each different HID, and that a patch written on one operating system platform should work in the same way on another. We have tried to preserve flexibility while simplifying the complicated

HID APIs as much as possible.

In this paper we present our current progress implementing [hidio]. We also discuss [input_notifier] external that monitors device plug/unplug events, that works in concert with [hidio] to allow a patch to be responsive to these events. We will discuss the design decisions that have led to our current implementation, and will point out our plans for future work on the HID toolkit.

2. PREVIOUS WORK

We are aware of a range of existing work, including some HID access tools outside the Pd and Max/MSP environments. However, Max's [hi] and Pd's [hid] are the most related to what the HID Toolkit is trying to accomplish, so we investigated these existing tools in more depth.

2.1 Max's [hi]

A number of objects exist within Max/MSP for getting data from HID's, such as [hi], [hidin] [13], [MouseState], [Insprock], [Wacom], [forcefeedback], and [MTCcentroid]. Each has a distinct programming interface. [hi] is a good example for coherent integration because it provides a single interface for getting input data from many different kinds of HID's. However, [hi] has limitations that impact its usefulness for creating and performing with new instruments. For instance, it only checks for new devices on startup, and blocks other instances of [hi] (and other programs) from accessing a connected device. Also, device elements like buttons and axes are identified with numeric tags rather than semantically meaningful categorical identifiers. Finally, [hi] does not support any HID output, making it incapable of controlling on-device feedback.

2.2 Pd's [hid]

Pd has a number of objects and patches for using HID's such as [MouseState], [linuxmouse], [linuxevent], [joystick], the Gem HID objects, and [hid] [15]. But, like the Max/MSP objects, these objects all have different interfaces, requiring a user to learn each object separately in order to use the device it supports. We feel that since a large range of HID's all share the same basic capabilities for input and output, the interface to access these objects should be unified. Although parts of the current project are firmly rooted in the existing [hid] object for Pd, a number of limitations of [hid] motivated our work: [hid] has no support for elements which appear more than once on a given device; it has very limited output support; no Windows support; and, like Max's [hi], multiple instances of [hid] 'steal' events from each other.

3. HID DESIGN ISSUES

3.1 HID APIs

A wide variety of available HID APIs exist, ranging from cross-platform to operating system specific, which some even being device-specific. We have researched and used many of the relevant APIs, finding strengths and weaknesses of each. Ultimately, we decided to use each operating system's primary API, which were the most flexible but also more difficult to use. For detailed information about the APIs that we considered, please visit our web page on the topic [5].

3.2 Managing Devices

A common programmer's frustration in handling user input is how to specify the device to open for communication. A number of schemes exist for identifying devices, each with its own advantages and disadvantages. The HID Toolkit supports several methods for selecting devices: device number, device type (gamepad, joystick, keyboard, etc.), and vendor ID/product ID. The device number is a unique numeric identifier assigned by the operating system. It is often useful behind the scenes in a selection menu, to select between multiple devices of the same type. However, often the specific model of the device is not as important as the general *type*. For example, a patch designed around a mouse and joystick may work acceptably with any mouse and joystick. In these cases, the device type would be an appropriate way to select the first 'mouse' and a 'joystick' found by the system. Vendor and product IDs and provide the most detailed device information, but typically in a less human-friendly format. These values can be used to select a particular device, even when there might be multiple devices of the same type plugged in. See figure 1 for more details.

[hidio] does not monitor HID plug/unplug events, instead it builds a device list just before it tries to open a device. The ability to monitor device insertion and removal can be useful though, to avoid having to restart a patch if a device is removed and reconnected, or to mute sound if a control device is unplugged. The [input_notifier] object was designed to communicate HID plug/unplug events. It is currently implemented on GNU/Linux, and [input_notifier] objects are created with a human-readable argument such as 'Microsoft Sidewinder Dual Strike' causing them to filter incoming messages for devices of that model. In the future, [input_notifier] will recognize identical ways of specifying devices as [hidio] so that the same message can be used to specify devices in both [hidio] and [input_notifier].

3.3 Device Polling

Most USB mice have a poll interval of 10ms [12]. While it is possible to get input reports more frequently, we believe that it would not be perceptible nor would it affect performance in a noticeable way. In certain specialized circumstances, such as accurately measuring human response times or implementing a PID loop, it could be useful to set the interval to a smaller time. We are currently exploring whether it is effective to set the poll interval to very low times when using output reports. This could allow the processing of haptic feedback to happen on the host computer rather than on the embedded microcontroller in haptic HID's. We use a default fixed poll interval of 5ms.

4. INTERFACE DESIGN ISSUES

4.1 Event Scheme

For [hidio], the labeling of the HID events was carefully designed for flexibility and human comprehension. The symbols should clearly represent the device's elements, adhering to existing schemes as much as possible. It is derived from the Linux input system, and USB HID [1], USB HID [4], and is used on all platforms. While the Linux scheme was generally well organized, some aspects are difficult to abstract. USB HID specifies a number of different kinds of absolute X

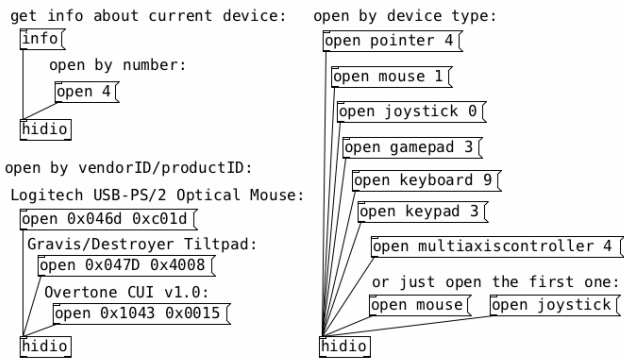


Figure 1: Multiple ways to select a HID with [hidio]

axes for different ‘usage pages’ like ‘Generic Desktop’, ‘Simulation’, ‘VR’, ‘Sports’, and ‘Game’. [hidio] uses the same event names for all related movements: ‘x’, ‘y’, ‘z’, ‘rx’, ‘ry’, ‘rz’,³ ‘slider’, ‘dial’, ‘wheel’, and ‘hatswitch’. For outputting data to control LEDs and force feedback, [hidio] sticks quite close the USB specifications.

For the button scheme we followed USB HID, which simply numbers the buttons. The Linux input system uses a different naming scheme for each device type, for example, `btn_left`, `btn_middle` for mice or `btn_trigger`, `btn_base` for joysticks. One key advantage of the numbering scheme is that it allows buttons on one device to work in patches written for other devices. A patch written for a joystick could be used by any other device with buttons and absolute axes, like a gamepad or tablet. A disadvantage is that the user might have to test the device to find the number scheme, rather than reading the label (‘button.0’ vs. ‘btn_trigger’).

All joysticks and gamepads have absolute X and Y axes, which are labeled as ‘absolute x’ and ‘absolute y’. This allows a patch built for a gamepad to be controlled by a joystick, and vice versa. The event labels aim to describe the physical motion of the human with the device; ‘absolute y’ means moving a physical element towards and away from oneself, while ‘absolute x’ means moving that same physical element from left to right. In addition, some devices report events using the ‘vendor defined’ codes, which by definition do not have names ascribed to them. These events are reported using the hex value as a symbol.

4.2 status outlet

As part of the push to handle as much as possible in the patching environment [hidio] can be queried, and the results processed within Pd/Max. An example status message is the minimum and maximum values a given HID element can report. A joystick’s X axis might output data between 0 and 127, and the value could be used to automatically scale the X axis data to between 0 and 1 in order to match most parameters (audio amplitude, OpenGL colors, etc.). Currently the user can get the following parameters from the status outlet: product and vendor IDs, manufacturer and product strings, data ranges for each element, the USB HID ‘Application Usage’, open status, poll interval in milliseconds, and transport bus.

³‘r’ stands for rotation here

4.3 Output to a HID

Using HID APIs, there are two ways of sending data to a device. One is using Force Feedback (FF), a standard that was designed allow gaming devices to present haptic feedback to the user. Another option is sending raw output commands directly to a device. The FF APIs are essentially a wrapper that send standard HID output commands, meaning that every FF-capable device can understand these HID commands. The more flexible method is to send HID output commands directly, since this permits control of more parameters, only limited by the capabilities of the device itself. The output commands are structured like the input reports. As with inputs, each device provides a list of supported output elements. In the current implementation (Max/MSP on Windows only) data can be sent to a device using a Max/Pd message with the same format as the input events are reported. [hidio] then builds an output report and sends it to the device. This includes an instance number in case there are several instances of the same usage page and usage id being present as output elements. We plan to use symbolic labels for the output elements, although [hidio] will also need to respond to numeric labels for the output labels in order to support the limitless number of “vendor defined” output element types.

5. MEASUREMENTS

In this section we discuss our investigations latency and jitter in computer-HID communication.

5.1 latency

In order to measure the latency induced by various design decisions, we set up some latency measurements on Mac OS X. We compared the timestamp on each event to the `mach_absolute_time()`, which is claimed to be accurate in the range of nanoseconds, or at least microseconds. Then, the last 8192 latency values were averaged, while excluding event latencies greater than 100 which would sometimes arise. Only x,y mouse movement and keyboard presses were measured.

We have noticed that there is actually quite a bit of jitter, and we plan to perform further jitter-related analysis. Since human performers can more easily compensate for latency than jitter, it may be useful to add latency in order to smooth out the jitter. We have not yet discovered a method for getting a timestamp for each event on Windows, so we do not yet know if this approach is possible on that platform.

5.2 HID echo test

In order to determine the speed at which data can be transmitted to and from a HID device a ‘Create USB Interface’ (CUI) [14] with a special loopback firmware was used. This firmware implements a device that has both input and output elements, and we configured the outputs to be directly connected to the inputs, meaning that every byte of data sent by PD to one of the outputs gets immediately reported as input. A test patch was created to send arbitrary integers at short intervals to the device, and the the echoed data was monitored for missing information. Transmit intervals of down to 12 ms succeeded, but with shorter intervals the echoed data began to show many missing values. The same test was repeated with a normal input / output firmware and a cable connecting output pins directly with

Table 1: average latency at a given poll time for two implementations

	1ms	3ms	5ms	10ms	25ms
threaded	4	4	4	4-5	10
threadless	4	4	4	4-5	10

input pins. This setup showed similar results to the loop-back firmware tests.

An adjustable parameter that affects latency is the ‘bInterval’, a value that each HID reports to the system to suggest a polling interval. We ran another test after lowering this suggested rate from 10 to 5 milliseconds, and found that we could then reduce the transmit interval to 7 ms before data was lost. This result suggests that maximum data rate depends on the bInterval setting within the device’s firmware, and may not be completely determined by the [hidio] object. This test was performed on a computer running Windows XP, and tests on Linux and OS X will be needed to understand whether they respond similarly.

6. CONCLUSION

In this paper we have presented our ongoing work to permit human interface device usage on Max/MSP and Pure Data. Our work aims to provide access to HID devices in a consistent, usable manner on GNU/Linux, Mac OS X, and Windows. Our primary goal in creating [hidio] and [input_notifier] is to encapsulate the overly-complicated HID API and to enable device plug/unplug monitoring, facilitating the usage of HID devices in interactive patches. We believe that a good interface to HID devices can be flexible and high-performance while still remaining accessible to novices. However, abstraction of complexity necessarily produces a tension between ease of use and flexibility, and we have attempted to find a satisfying balance. We expect the HID Toolkit to reach a broad audience with a wide range of technical sophistication, from expert designers of electronic instruments to students taking their first physical computing course. We hope that the existence of the HID Toolkit expands the set of possibilities for these audiences in their creative work at the intersection of art and technology.

7. FUTURE WORK

In the future, we plan to undertake a more rigorous analysis of latency data from various HID devices, in order to understand better the causes of jitter and how best to mitigate this problem. Bluetooth connectivity is another exciting feature that we hope to enable, since the growing number of Bluetooth HID devices provide a convenient way to make a musical device wireless. Finally, with the help of the growing user communities for Pure Data and Max/MSP, we plan to expand our growing library of ‘wrapper’ abstractions that encapsulate the basic [hidio] object to work with a wider range of high-level device classes, as well as with specific devices that require special data handling.

8. ACKNOWLEDGEMENTS

We thank the user and developer communities of Pure Data and Max/MSP for their ongoing input and assistance throughout our development.

9. REFERENCES

- [1] Hid information. <http://www.usb.org/developers/hidpage/>.
- [2] Interactive Studio Seminar 2. <http://idmi.poly.edu/ms>.
- [3] Principles of Electronic Music Controllers. <http://www.media.mit.edu/resenv/classes/MAS960/>.
- [4] Usb pid. http://www.usb.org/developers/devclass_docs/pid1_01.pdf.
- [5] Working with USB HID devices. <http://at.or.at/hans/research/nime/hid/>.
- [6] A. Black. El Lechero. <http://aug.ment.org/lechero/>.
- [7] G. D’Arcangelo. Creating a context for musical innovation: a nime curriculum. *Proceedings of the 2002 conference on New Interfaces for Musical Expression (NIME’02)*, 2002.
- [8] R. L. DuBois. An Interview with Luke DuBois. <http://cycling74.com/community/lukeadubois.html>.
- [9] S. Greenberg and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. *Proceedings of the 14th annual ACM symposium on User Interface Software and Technology (UIST’01)*, pages 209–218, 2001.
- [10] L. Gruenbaum. Samchillian Tip Tip Tip Cheepeepee. <http://samchillian.com>.
- [11] L. Kessous. Bi-manual mapping experimentation, with angular fundamental frequency control and sound color navigation. In *Proceedings of the 2002 conference on New Interfaces for Musical Expression (NIME’02)*, Dublin, Ireland, 2002.
- [12] krejler. Increase usb mouse polling interval. <http://www.linux-gamers.net/modules/wiwimod/index.php?page=HOWTO+USBPolling>.
- [13] O. Matthes. hidin object. <http://akustische-kunst.org/maxmsp/dev/>.
- [14] D. Overholt. Musical interaction design with the create usb interface: Teaching hci with cuis instead of guis. In *the proceedings of the International Computer Music Conference*, 2006.
- [15] H.-C. Steiner. [hid] toolkit: a unified framework for instrument design. In *Proceedings of the 2005 conference on New Interfaces for Musical Expression (NIME’05)*, Vancouver, BC, Canada, 2005.
- [16] B. Verplank, C. Sapp, and M. Mathews. A course on controllers. *Proceedings of the 2001 conference on New Interfaces for Musical Expression (NIME’01)*, 2001.