HID Packet Parser Documentation

This document describes how Mixxx 1.11 javascript implementation of HID packet parsers works. Our goal is to help mapping writers undrstand why and how things happen.

As usual, this document is just a draft and needs more content.

For details about HID data formats and concepts, please read details from following web page:

http://www.usb.org/developers/hidpage

Also please remember that javascript does not have classes: you can write functions which look like classes in other languages, with member functions and private attributes, but they are not classes.

To help bashing the myth of javascript classes, we call all these prototypes below as functions.

Short HID Format Summary

When working with HID, forget anything you have learned about MIDI. Unlike MIDI, HID sends *data packets* with size anything from 1 byte to *maximum value allowed by standard (check from specs what it is),* and similarly arbitrary size HID packets can be sent to devices.

In HID, each input and output packet usually contains information for multiple fields: for example, a HID keyboard sends 1 byte keycode for 6 different pressed keys simultaneously with three byte header with values 0x1,0x0,0x0 in the header, so total packet size is 9 bytes. Similarly, on a typical HID controller, you will receive a packet with multiple control values and send data for multiple output controls (LEDs, text fields etc) in one larger packet.

The HID standard has concept of *HID usage page*, which contains descriptions of each byte in input and output packets. Unfortunately it's not mandatory to provide these packets, and typical DJ controller does not provide these packets. Thus, we must parse our own HID packets in mixxx.

Unlike MIDI data, HID can contain multiple byte values for single field: for example, most HID controllers send fader details in a 2 byte (*short)* numeric field, not necessarily using whole range of values. As an example, Pioneer CDJ pitch fader has value range -1000 to 1000, expressed with two bytes. Another example of complex HID packet fields is the output packet for Pioneer CDJ devices, which sends the *waveform display* in one packet to the device: the packet size is hundreds of bytes.

Supported *HID input* field types

The current mixxx javascript implementation for HID input packets supports following types of fields in the parser:

- One byte signed or unsigned numeric value (packing codes *b* and *B*)
- Two byte signed and unsgined numeric value (packing codes *h* and *H*)
- Four byte signed and unsigned numeric value (packing codes *i and I)*
- Parsing of *bit mask* fields from one packet of above sizes

The bit mask fields are implemented as normal numeric value field, which has type *bitvector* and which has value of internal class *HIDBitVector*. Any bit mask size from 1 to number of bytes in the

field is supported, but usually you want to address one bit for input button toggles.

Note that you can't choose the endianess of the packing in current code. Also note the javascript numbers implementation for bit arithmetic limits the size of one field to 32 bits.

Supported *HID output* field types

For HID packet output, I could not bother writing any other types of output fields than bit masked bits in one byte size packets and sending of one byte numeric values, signed or unsigned: main reason I did not implement other types of fields yet is that I don't have any devices which need this. It is easy to add other packing formats, when required. Longer fields than one byte for output need to be constructed by script writer for now.

Packet signature

Each HID packet has following attributes:

- *Packet name* is used to refer to the packets from javascript and are not sent to controller. In most cases, the device sends one *control input status* packet to indicate state of buttons and controls, and this packet should be named *control* to allow the parser automatically parse it.
- *Packet header* can contain static bytes in the beginning of the packet. This is completely dependent on the controller, for example Pioneer CDJ control packet does not have any header, but output packets have 1-4 byte prefixes. The header is given as javascript array, or empty array if no header bytes are set. The header bytes are always sent in the packet and the received packets are recognized by the header signature.
- Optional *packet callback* function can be registered in packet declaration for input packets. If a packet callback is defined and the packet is received, the packet is sent to the callback function after field values are parsed, without calling any packet field parsing functions.

Internally, each packet registers the packet fields following mixxx naming conventions, i.e. each field must have a *group* and *name.* The group and name don't need to be valid mixxx controls, but if the group is valid mixxx control group, we try to automate mapping the field to mixxx controls.

Packet Field packing and unpacking

Since HID controller packet fields can be larger than one byte, I thought it good idea to implement field description based on following details:

- *Group* defines the group name for the field. The group can be any string, but if it matches a valid mixxx control group name, it is possible to map a field to a control or output without any additional code. See recommendations for custom group naming later in the document to undestand some HIDController specific goodies.
- *Name* is the control name for the field. The name can be any string, but if it matches a valid mixxx control name in the group defined for field, the system attempts to attach it directly to the correct field. Together *group* and *name* form ID for the field (group.name).
- *Offset* defines field's offset from the start of the packet. For example, with 2 byte header, first field in packet must have offset of 2, and if the first field was packing h or H (a 2 byte short integer), second field should have offset 4.
- *Pack* is one of the field packing types mentioned above. Purpose of this is to allow script writer to receive or send the correct numeric values for a field, not bothering with end of

field or parsing the received value itself

- *Bitmask* is only defined for fields which are not expected to handle all bits in the control field. For fields with bitmasks, you can define same offset and pack multiple times with different bitmask values to get for example all 8 bits of a buttons state byte to different control fields in *addControl* input packet command. Masking multiple bits should work but has not been as widely tested.
- *Callback* is optional argument to bind a callback function to the field in input packets. If a field has callback, the function is called with parsed field value, and any calls to other standard functions are skipped. It's not recommended to bind callbacks to function while describing the HID packets, because it makes the packets less reusable: you can bind a callback to fields later in your script with HIDController.RegisterCallback function.
- *Is_encoder* is boolean field, undefined by default, which makes the field value as encoder type field. Encoder type fields take the range of values available in the field, calls the field functions with correct -1 or 1 offset based on direction, and wraps the number when max to 0 or 0 to max changes are seen.

  Note: scaling functions and automated field group resolutio are **NOT done** for fields which have a registered callback function. The callback function is expected to do required field name resolution and scale values itself.

Scaling of field values to mixxx ranges

In almost every case, a HID controller sends data values with input fields which are not directly suitable for mixxx control values. To solve this issue, HIDController contains function to scale the input value to suitable range automatically before calling any field processing functions.

Scalers can be registered with *HIDController.registerScalingFunction(group,name,callback)* in HIDController.

HIDPacket function

HIDPacket function wraps all details of reading HID input packets and sending HID output packets to devices. HIDPackets are expected to be defined by the *HID description* writers only, people using the HID device description only need to read the assigned field names.

HIDPacket packets have some common functions to use:

- *HIDPacket.AddControl(group,name,offset,pack,bitmask,callback,is_encoder);*
  This function is used to add a normal field to packet, as described above. Same function is used to add numeric value fields (bitmask is *undefined)* and bit mask field offsets (*bitmask* has a valid value).

  The last parameter *is_encoder* is a boolean value to define how the field values are processed. See details above.
- HIDPacket.AddLED(group,name,offset,pack,bitmask);
  This function is used to define an output LED control to output packets: LEDs are specialized fields to allow us to define LED colors nicely and to add API functions like HIDController.setLED("[Channel1]","play","green") easily.
- *HIDPacket.setMinDelta(group,name,minimum_change);*
  Some HID controllers seem to send lots of minimal changes which are not meaningful for

mixxx users. This allows you to set a threshold value, which ignores the input if field value changed by less than given amount.

- *HIDPacket.setIgnored(group,name);*
  This function can be set in script code to ignore a field you don't want to be processed but still wanted to define to make packet format complete from specifications. In most cases you can just leave the ignored field undefined in your mapping.

Note that all field types, the internal structure is similar. In case of *bitmasked fields*, the field type is set to *bitvector* and the bit values are stored to *HIDBitVector* stored in field value. Similarly a LED has type *led* and some extra attributes for blinking etc.

HIDController Function

HIDController is a javascript prototype function, with multiple calls allowing mostly automatic HID packet processing. It is expected to be used in place of the *new Controller* calls in normal MIDI mappings.

This function prototype is quite complex and it's not yet documented fully here: please read source in common-hid-packet-parser.js, which contains nice comments for each function and what it does.

It also defines multiple documented attributes, which for example allow implementing scratching just by adding special field names 'jog_touch' and 'jog' to the HID fields, and adjusting the attribute values for scratching to get correct performance.

Required end-user script functions

Following functions need to be implemented in end user classes. This is due to issues with mixxx qtscript calls and namespaces, where *this.functionName* does not work directly from a prototype function like HIDController. Same rules as for MIDI scripts and HID scripts without HIDController prototypes apply.

- *init();*
- *shutdown();*
- *incomingData(data,length);*

See below examples for common usage of these functions.

Minimal example scripts using HID packet parser

There are multiple very simple mappings in mixxx 1.11 tree, which can be studied for the expected ways of using the common hid packet parser code.

The simple mappings are in:

- *common-hid-devices.js*
  contains example descriptions for standard HID trackpad/mouse and HID keyboard
- *Apple Bluetooth Keyboard.cntrlr.xml,* Apple Bluetooth Keyboard.js
  Example script just dumping the key presses received from a standard USB keyboard
- Sony SixxAxis.cntrlr.xml
  Simple mapping using separate files for HID packet declaration and script mappings
- Sony SixxAxis.hid.js

The hid packet declarations for previous XML

- Sony SixxAxis.js
  Actual mixxx control mappings for PS3 controller inputs, using previous hid.js file
- Nintendo Wiimote.*
  Similar files as sixxaxis mappings, introducing the style of writing abstract packet format and linking everything in actual script to controls.

If you wish to understand how fields in HID are mapped to controls, please read file *Nintendo Wiimote.hid.js*. For more complex usage, but older and not as clean code, you can read *EKS Otus.js* which was the first mapping written with these classes (and will be rewritten in style of the Wiimote controller scripts).