**Input Notes**

It's very likely you forgot about it, so don't worry: a Usage is some control on the controller, and the ones we're esp. concerned with are the valued usages.

**Basics of HID calls**

HID calls will return HIDP\_STATUS\_SUCCESS if things worked.

**Button State - HidP\_GetUsages**

HidP\_GetUsages fills in usage array w/ numbers to buttons that are pressed - upon polling, zero out a button state array, and loop through the usage numbers. You must subtract buttonCaps->Range.UsageMin from the usage number to get the specific button number on the joystick!

**Valued Controls - HidP\_GetUsageValue**

To get valued controls, call HidP\_GetUsageValue. You can only pull one axis at a time to a value var with this call, so you need to loop through the cap count in Caps->NumberInputValueCaps given by HidP\_GetValueCaps. To figure out what axis is being referred to, you'll need to check the cap's usage range; if it affects a range, that implies the cap's referring to a specific joystick control, and you need to iterate through the range, updating each axis as you go. To get all the value caps, call HidP\_GetValueCaps and store this data in another array on device build. Once you have that, the physical caps for a usage are in ValueCaps.PhysicalMin & ValueCaps.PhysicalMax. This all implies some sort of organization of each controller into arrays of control classes, such as buttons and axii. Before you can do that, you need to write some tests to enumerate the usages of a HID.

Your test rig is the Saitek X52 Pro, easily the most complicated joystick readily available on the market. If you can do this, you can handle any joystick.

**Input Namespace**

The input namespace handles everything regarding controllers; from here you can get the current list of controllers, get a specific controller, and a whole bunch of other junk.

**Controller Class**

Individual controllers are represented by the Controller class.

Controllers have:

* a platform specific ID
* a type:
  + Keyboard
  + Mouse
  + HID
* an array of controls, represented either as bools (for digital buttons) or U16s (for analog controls):
  + Buttons are stored as a U64 bitfield (since some psycho decided the X52 needed 39 buttons); each button gets its own flag. To get/set, AND/OR against 1 << (button num).
  + Values are stored as an array of ControlAxis, usually a static array of 16. A ControlAxis consists of:
    - U32 magnitude
    - U32 indicating max value (min value seems to universally be 0)
  + From the few controllers used, axis seem to be unique; there isn't more than one X,Y,Rz, etc. Given this, we could index by hashed usage page and usage. To access standard axii, get the hash of common axii (say 0x1:0x30 for joystick X axis).
    - Alternately, just lay them out as is, figure out the configuration later. The axii will need to be scaled by their min and max, however. To do so, subtract the middle value (max / 2) from the control value, and divide by the middle value.

Analog's stored as U16 rather than F32 because many joysticks store their values in [0 - 65535], where 32768 (0x8000) represents the control's center. You can get a [-1.0 - 1.0] value w/ a convienence function. Generally, a controller will have state for this and the previous frame, to allow tracking of button holds and presses.

**Platform Specifics**

Under Windows, we first enumerate the controllers we're interested in via Raw Input. That API uses values from the USB HID Usage Tables (UsgTabs), if that's not in the docs in this project folder, you'll need to google it.

**Enumerating Controllers**

To get the number of attached controllers, call GetRawInputDeviceList and pass a U32 to puiNumDevices. From there, you can make an array of RAWINPUTDEVICELIST; pass that, the U32 you caught, and sizeof(RAWINPUTDEVICELIST) to GetRawInputDeviceList and the list will now be filled w/ device info. RAWINPUTDEVICELIST contains:

* device handle
* device type, can be:
  + RIM\_TYPEKEYBOARD
  + RIM\_TYPEMOUSE
  + RIM\_TYPEHID for anything else.
    - "anything else" can be literally anything, not just joysticks and gamepads.

To get info on a device, call GetRawInputDeviceInfo. It takes:

* device handle
* command, can be:
  + RIDI\_DEVICENAME to get the device name (NOT human readable, that's a whole other can of worms); if you pass this, the func will return the string ptr in pData and the string len in pcbSize.
  + RIDI\_DEVICEINFO to get device state; with keyboards and mice this can be directly accessed for button/mouse wheel state.
  + RIDI\_PREPARSEDDATA to get previously parsed data. This is generally what you want for joysticks; You will be given data that can be cast to PHIDP\_PREPARSED\_DATA.
* a void pointer pointing to a buffer for the data; pass NULL to just get the required size for the buffer.
* a pointer to a U32 indicating the buffer size. If the device is not a HID, you can immediately get info on the number of buttons or keys probably. If it IS an HID, you will need to use HidP\_\* functions that are in hid.dll or directly linked from the .lib files and headers from the Windows Driver Dev Kit (WDK).

*Notes on HIDP\_\**

These functions return an error code. On success, these functions will return HIDP\_STATUS\_SUCCESS.

*DLL or LIB?*

For compatibility with other versions of windows, you can either load a DLL or link in the WDK and include hidsdi.h, hidusage.h, and hidpi.h. The latter will require installing the 620MB WDK to get the libs, but whatever. The main question is whether or not this will be compatible with other OSes besides Windows 7. If you load a DLL from the local system, you can be pretty confident it'll be compatible.

To get specific info on an HID's digital & analog controls, you need to get the current general control info; call HidP\_GetCaps, passing the preparsed data you got and a HIDP\_CAPS pointer. The caps struct will now contain:

* The number of digital controls (NumberInputButtonCaps)
* The number of analog controls (NumberInputValueCaps)
* The usage and usage page (though you should already have this stored elsewhere)
* The size of an input report (InputReportByteLength)

among MANY other things, but these input structs are what matters most. Digital control caps **will** be in an array of HIDP\_BUTTON\_CAPS of length NumberInputButtonCaps, and analog caps will be in a corresponding array of length NumberInputValueCaps. Once those arrays are alloc'd, call HidP\_GetButtonCaps for digital controls, HidP\_GetValueCaps for analogs, passing HidP\_Input for the report type. Most of what you care about is now in a HIDP\_VALUE\_CAPS struct, which has:

* Usage page (what control type) (UsagePage)
* Iff a range (IsRange == true), a Min and Max usage to indicate a specific axis on the control (UsageMin, UsageMax).
* Info's unclear, but each cap may represent a specific control, such as a joystick on the Saitek X52, so the cap have a usage range for each axis the control operates over.
* Min and max vals for the control's range (LogicalMin, LogicalMax). Note that according to UsgTabs these sometimes aren't used by joystick drivers! If the controller is a joystick, feel free to assume the range is [0, 32K, 64K].
* Finally, both value and button caps have a .NotRange.DataIndex/.Range. DataIndexMin/Max field that contains an ID that can be used to directly address a given usage. For you, it's not necessarily very important, however.

You now know about all the input controls on the controller! Organize this info as needed so you

don't need to get this info on every WM\_INPUT.

*Other Stuff*

* The \*\_CAPS structures can also contain string identifiers in each capability.
* Buttons are typically stored in ONE capability. All buttons are on page 0x09, and their specific usage indicates their button number (the whole page is for just buttons) - see UsgTabs, p.67.

**Registering Controller Types**

To indicate a device type, we pass a RAWINPUTDEVICE struct to RegisterRawInputDevices. The struct holds 4 things in particular here:

* Usage Page - 0x01 for us (Desktop Page)
* Usage - 0x02 (Mouse), 0x06 (Keyboard), 0x04 (Joystick) or 0x05 (Gamepad). Most likely, you won't poll for mice or keyboards, and just use info in the WM\_INPUT messages for them.
* Flags - usually 0x0 until further notice
* Window Handle - 0x0 unless you have a window open, then it's that window's handle. Fortunately, Win32Platform keeps the handle to the current game window. Pass all desired device types in an array to RegisterRawInputDevices; if it returns false, call GetLastError to find out what happened, otherwise you're ready.

**Getting Controller State**

When a registered controller changes state, you'll get a WM\_INPUT message. WM\_INPUT contains:

* The input data's handle in msg.lParam. Note that this is NOT the same as the device's handle, but the struct's header does contain the device handle!
* The input code in msg.wParam. Can be:
  + RIM\_INPUT - event happened while window was in foreground
  + RIM\_INPUTSINK - event happened while window was NOT in foreground.

The exact data buffer is unknown, you first need to get the size. Call GetRawInputData w/:

* the handle to WM\_INPUT's RAWINPUT struct
* RID\_INPUT
* NULL
* pointer to U32 size var
* sizeof(RAWINPUTHEADER)

Now you can alloc a buffer; alternately, you might keep one buffer large enough for the state on any controller. You'll need to experiment for that data. Anyway, now that you have the buffer, call GRID as before, but also pass the buffer as a PRAWINPUT. This can then be interpreted as needed; if the handle indicates a mouse or keyboard, cast to RAWMOUSE/KEYBOARD and read accordingly. Otherwise, cast to RAWHID and parse.

**Parsing a RAWHID**

Given a RAWHID, the state will be in .hid.bRawData. You need to pass the preparsed data you found previously to HidP\_\* functions to interpret the data. To get the state of the digital controls, call HidP\_GetUsages w/:

* HidP\_Input (indicating input report)
* usage ID of your button caps
* NULL (get data for all buttons)
* array of USAGE structs (button state)
* number of buttons
* preparsed data
* the raw data as a PCHAR
* the raw data's size (.hid.dwSizeHid)

This will fill the USAGE array with all pressed buttons. Subtract the button cap's min usage to get the specific button pressed. To get the state of a SINGLE analog control (an axis usually), call HidP\_GetUsageValue w/:

* HidP\_Input
* usage ID of the specific control
* NULL (get top-level data)
* the USAGE struct corresponding to the specific control
* a PULONG to store the control's value
* preparsed data
* the raw data as a PCHAR
* the raw data's size (data.hid.dwSizeHid)

As previously noted, you'll still need to reinterpret the value returned in terms of the specific control's range. Also remember that each joystick/throttle on the controller might have multiple usages, and so you may need to retrieve over a range! (A joystick is a 2D control, after all.) The min will likely be somewhere around 0x30-0x32 (X,Y,and Z), no telling where the max is.

**HIDs**

Largely handled according to prior sections, but there's one small problem: how are multiple button ranges/any value ranges handled? They're currently not in the code.

*Dead Zones*

Dead zones can't always be handled by individual axii; most of the filters you care about require two axii being passed. Other than that, though, you're good.

**Keyboards**

Although the keyboard can list state for up to 256 different keys, there's nothing indicating what each key **means**. Under Windows, alphanumerics are same as in ASCII (remember that there's no distinction between capital letters; alphabetics use capital codes and numerics use numeral codes). Several other keys also correspond (as far as an ASCII code, anyway):

* Space
* Backspace
* Return
* Tab (as ASCII "Horizontal Tab")

For all else, it's platform dependent. The max number of keys in a keyboard standard is somewhere around 124 (an old Gateway standard) before counting special keys like the Windows key (or that friggin' menu key to the right of the right Alt...), which brings it closer to 126. Currently, the Keyboard class uses 256 bits to store current state; in theory this could be cut in half, but virtual keys beyond ASCII 127 (0x7F) must be remapped. This alone means you'd need to call an extra remap on every keypress, but it'd be necessary for cross-OS configuration anyway.

Anyway, if 128 bits were used instead, you'd have 128 - (10+26+4) = 88 free bits. There's WAY more than that listed in Windows's VKey list (something like 190 keys), but some of those keys can be ignored:

* the 20 or so media keys
* the ~15 IME keys? Not sure when those get used on say, a Japanese keyboard
* the 5 or so mouse keys (weird, but makes its own kind of sense)
* the 12 function keys above F12 if you don't really care about those old Gateway keyboards

Excluding all those, that leaves ~190 - (10+26+4+20+~15+~5+12) = ~98 keys unaccounted for. Still not enough, but workable. The lookup table would be simple, an array of 128 U8s - the array would just be platform-dependent.

Currently, keycode 255 indicates an invalid key – use this for unset keybindings.

**Mouse**

Largely tested; need to test position system, though.