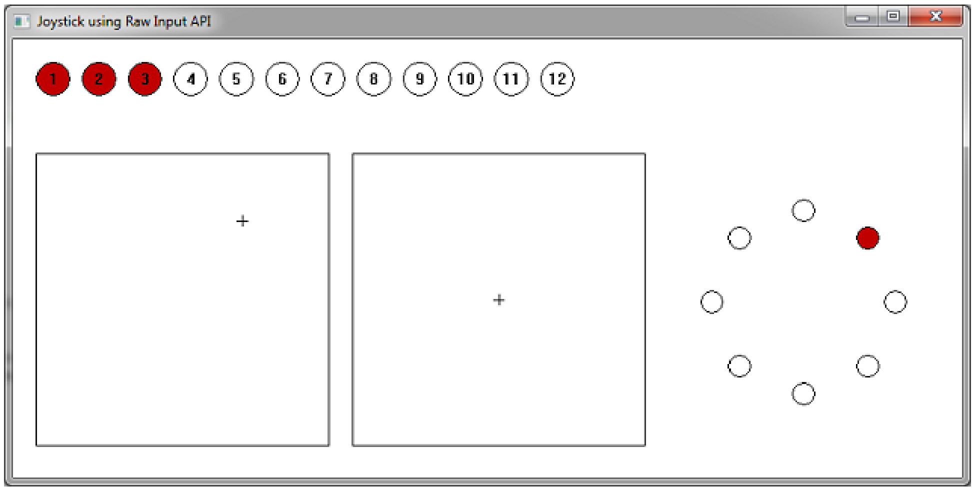
Using the Raw Input API to Process Joystick Input

By **Alexander Böcken**, 23 Apr 2011

★★★★★ 4.60 (4 votes)

Download source - 9.07 KB

Download demo project - 5.44 KB



Introduction

Microsoft discourages the use of DirectInput for keyboard and mouse input in games, but it is still recommended to process data from a legacy joystick, or other game controller. New applications should use the Raw Input API, i.e., to take advantage of computer mice generating data at 800 DPI or even more. Additionally, Microsoft introduced XInput to allow applications to receive input from the Xbox 360 Controller. Hence, game developers have to cope with up to three different APIs when designing their input system.

In this article, I will show how to use the Raw Input API to process joystick/gamepad data, making at least the use of `DirectInput` obsolete.

Background

With Windows XP, Microsoft introduced the Raw Input API to support other Human Interface Devices (HIDs) than the traditional keyboard and mouse. An application can use this API to get data from any HID one can imagine. But aside from keyboard and mouse input, the API exposes no structures or functions to interpret the data coming from the devices, thus making it useless to game applications that want to make use of the vast number of different joysticks and gamepads available. The Raw Input API documentation does not mention how to retrieve such valuable information as the number of buttons, the number of axes, the existence of a hat switch, and so on. (I guess this is why it's called "raw" ;-))

In the effort of developing a game engine, I wanted to add support for my Logitech RumblePad 2. For mouse and keyboard input, I have already used the Raw Input API, so it seemed logical to look at it first before trying to implement a second branch of code that manages `DirectInput` and its device objects. In the end, I came up with a solution using the Raw Input API and the `HIDClass` driver (see Human

About Article

A basic way to interpret joystick data received from Raw Input API

Type	**Article**
Licence	**CPOL**
First Posted	**23 Apr 2011**
Views	**24,066**
Downloads	**4,083**
Bookmarked	**18 times**

C Windows Win32 Advanced Game

Print Email

Top News

Skills that self-taught computer programmers lack

Get the Insider News free each morning.

Related Videos

Learn NowOnline .com microsoft library

Learn NowOnline .com microsoft library

Related Articles

Using Raw Input from C# to handle multiple keyboards

Minimal Key Logger Using RAWINPUT

Input Devices in the Windows Driver Kit).

The next few sections of this article will show a basic way of retrieving raw joystick data and how to interpret it.

Before Using the Code

As already mentioned, the sample application works closely with the `HIDClass` driver. To compile the code, you must download the Windows Driver Kit (WDK) from the Microsoft website and set the project paths accordingly. You will need to link to *hid.lib* and if you get a whole lot of compile errors, you may have forgotten to include the "*\inc\crt*" path.

Step 1: Register for Joystick Devices

The first thing every application using Raw Input API does is to register for the kind of devices it is interested in getting data from.

```
case WM_CREATE:
    {
        RAWINPUTDEVICE rid;

        rid.usUsagePage = 1;
        rid.usUsage     = 4;        rid.dwFlags     = 0;
        rid.hwndTarget  = hWnd;

        if(!RegisterRawInputDevices(&rid, 1, sizeof(RAWINPUTDEVICE)))
            return -1;
    }
    return 0;
```

This is the most simple part. For joystick devices, we only need to set `usUsagePage = 1` and `usUsage = 4`. The terms Usage Page and Usage have their origin in the HID Usage Tables of the USB Implementers' Forum. They specify the various types of input devices and their controls, i.e., 1 is the id of the Generic Desktop Controls Page and 4 is the id of the Joystick Usage Name. In addition, to receive the `WM_INPUT` message, we set `hwndTarget` to the handle of our window.

Step 2: Retrieve the Device Data

In the `WM_INPUT` case, we obtain a pointer to the `RAWINPUT` structure containing the joystick's data.

```
case WM_INPUT:
    {
        PRAWINPUT pRawInput;
        UINT      bufferSize;
        HANDLE    hHeap;

        GetRawInputData((HRAWINPUT)lParam, RID_INPUT, NULL,
                &bufferSize, sizeof(RAWINPUTHEADER));

        hHeap     = GetProcessHeap();
        pRawInput = (PRAWINPUT)HeapAlloc(hHeap, 0, bufferSize);
        if(!pRawInput)
            return 0;

        GetRawInputData((HRAWINPUT)lParam, RID_INPUT,
                pRawInput, &bufferSize, sizeof(RAWINPUTHEADER));
        ParseRawInput(pRawInput);

        HeapFree(hHeap, 0, pRawInput);
    }
    return 0;
```

`GetRawInputData` converts the handle given by the `WM_INPUT` message to a `RAWINPUT` pointer. We then pass this pointer to `ParseRawInput` which does the bulk of the processing.

The `RAWINPUT` structure provides us with the device handle, type of input and the input data itself. For mouse and keyboard input, we can use the `RAWMOUSE` and `RAWKEYBOARD` structures, but for any other device we have to use the `RAWHID` structure which just contains the raw input data, as an array of bytes. It is now up to us to interpret this array.

Step 3: Interpret the Device Data

Interfacing with a Joystick using C#
Radio Control Transmitter - Joystick
Using multiple keyboards with different layouts on the same machine
Redirecting an arbitrary Console's Input/Output
Joystick Driver Project
Speech Recognition for the Web
DLL Source For Buffer-based MP3 Decoding
How to redirect Standard Input/Output of an application
17 Channel Logic Analyzer
SSIS Overview - Part I
Using the RawInput API to Process MultiTouch Digitizers
Introduction to PayPal for C# - ASP.NET developers
DLL To Decode MP3 To WAV/PCM
A Simple Snake Game: Engerek
Handwriting Recognition using Kernel Discriminant Analysis
Minimal Key Logger Using RAWINPUT
Managing Processes and Shell API Functionality using .NET, Part 1
Windows Mobile Remote Controller

At this point, the first important observation is that this array actually contains the state of our joystick (i.e., my RumblePad). If we press a button or move one of the axes, we get a new WM_INPUT message and an array that reflects the new state of the joystick. That way, we could disassemble the array and determine the bits corresponding to the buttons and bytes corresponding to the axes. But this works only for our special joystick and not for most of the devices out there. So, how do we interpret the data at runtime?

The Raw Input API gives us access to some device information using the GetRawInputDeviceInfo function. Given the handle to the raw input device and the RIDI_PREPARSEDDATA command we can obtain so called "preparsed data". The WDK documentation just says, "The internal structure of a _HIDP_PREPARSED_DATA structure is reserved for internal system use". Here is the code to obtain this data block:

```
CHECK( GetRawInputDeviceInfo(pRawInput->header.hDevice,
        RIDI_PREPARSEDDATA, NULL, &bufferSize) == 0 );
CHECK( pPreparsedData = (PHIDP_PREPARSED_DATA)HeapAlloc(hHeap, 0, bufferSize) );
CHECK( (int)GetRawInputDeviceInfo(pRawInput->header.hDevice,
        RIDI_PREPARSEDDATA, pPreparsedData, &bufferSize) >= 0 );
```

The first call to GetRawInputDeviceInfo just gives us the size of the preparsed data. We use that to allocate a block that fits the data and retrieve it in a second call to GetRawInputDeviceInfo. CHECK is a small macro that I use to handle errors. If the expression in the argument evaluates to FALSE or 0, the macro frees any allocated memory and returns from the function.

Now here is the big picture. The device data we get with every WM_INPUT message is actually a series of HID reports from the HIDClass driver and we can use the preparsed data to unpack HID reports and determine the pressed buttons and axis values. (I guess, this is the way DirectInput has been implemented.)

First, we determine the number of buttons the joystick has. For that, we use the HidP_* functions from *hid.dll*. Obviously, *hid.dll* is the user space library used to communicate with the HIDClass driver and the lowest level library Windows offers for that kind of I/O.

```
CHECK( HidP_GetCaps(pPreparsedData, &Caps) == HIDP_STATUS_SUCCESS )
CHECK( pButtonCaps = (PHIDP_BUTTON_CAPS)HeapAlloc
        (hHeap, 0, sizeof(HIDP_BUTTON_CAPS) * Caps.NumberInputButtonCaps) );

capsLength = Caps.NumberInputButtonCaps;
CHECK( HidP_GetButtonCaps(HidP_Input, pButtonCaps,
        &capsLength, pPreparsedData) == HIDP_STATUS_SUCCESS )
g_NumberOfButtons = pButtonCaps->Range.UsageMax - pButtonCaps->Range.UsageMin + 1;
```

HidP_GetCaps gets us the number of the different capabilities we can query, the Usage and Usage Page of the device and the length in bytes of HID reports. To query the number of input buttons, we allocate a buffer that fits multiple HIDP_BUTTON_CAPS structures and call HidP_GetButtonCaps to fill it. The first parameter of HidP_GetButtonCaps specifies the report type. HIDs can have input reports (i.e. for buttons, knobs, faders, touch screens, etc.) and output reports (i.e. for LEDs, displays, force feedback, etc). The next three parameters are the pointer to our allocated buffer, its length and the pointer to the device's preparsed data.

HidP_GetButtonCaps returns the capabilities for every type of HID control of our joystick, that is, its Usage. pButtonCaps[0].UsagePage indicates the usage of the first set of HID controls (see Table 1 of HID Usage Tables). UsageMin and UsageMax indicate the lower and upper bound of usage range, i.e. the range of indicies that is returned by the HIDClass driver for the buttons of the joystick. The number of buttons is then the difference of UsageMin and UsageMax plus one.

The reason why HidP_GetButtonCaps returns an array of HIDP_BUTTON_CAPS structures is that a joystick can assign different usages for different buttons. For example, my RumblePad has a button labeled "Vibration" that enables or disables force feedback effects. The UsagePage member of the first array element indicates Button Page for the twelve action buttons, whereas the second array elements indicates vendor-defined usage for the "Mode" and "Vibration" buttons.

We now get the value capability array. This array specifies the capabilities of HID controls that can have more than two states (i.e. pressed or released). These controls often have a range of values. For example, my RumblePad has two analog sticks, that is, four axes, each of which has a range of 0x00 to 0xFF where 0x80 equals the centered position of the stick.

```
CHECK( pValueCaps = (PHIDP_VALUE_CAPS)HeapAlloc
        (hHeap, 0, sizeof(HIDP_VALUE_CAPS) * Caps.NumberInputValueCaps) );
capsLength = Caps.NumberInputValueCaps;
CHECK( HidP_GetValueCaps(HidP_Input, pValueCaps,
        &capsLength, pPreparsedData) == HIDP_STATUS_SUCCESS )
```

The member UsagePage of the HIDP_VALUE_CAPS structure again indicates the usage of the value (i.e. which axis). PhysicalMin and PhysicalMax indicate the range of the value. Note that these are not used by the sample code provides with this article. The sample code assumes the range to be between 0x00 and 0xFF.

Now we obtain the state of the action buttons from the HID input report in pRawInput->data.hid.bRawData.

```
usageLength = g_NumberOfButtons;
CHECK(
    HidP_GetUsages(
        HidP_Input, pButtonCaps->UsagePage, 0, usage,
        &usageLength, pPreparsedData,
        (PCHAR)pRawInput->data.hid.bRawData, pRawInput->data.hid.dwSizeHid
    ) == HIDP_STATUS_SUCCESS );

ZeroMemory(bButtonStates, sizeof(bButtonStates));
for(i = 0; i < usageLength; i++)
    bButtonStates[usage[i] - pButtonCaps->Range.UsageMin] = TRUE;
```

HidP_GetUsages returns only those buttons in usage which are actually pressed. I guess that this code snippet is self-explaining.

Finally, we obtain the state of the valued controls from the HID input report.

```
for(i = 0; i < Caps.NumberInputValueCaps; i++)
{
    CHECK(
        HidP_GetUsageValue(
            HidP_Input, pValueCaps[i].UsagePage, 0,
                pValueCaps[i].Range.UsageMin, &value, pPreparsedData,
            (PCHAR)pRawInput->data.hid.bRawData, pRawInput->data.hid.dwSizeHid
        ) == HIDP_STATUS_SUCCESS );

    switch(pValueCaps[i].Range.UsageMin)
    {
    case 0x30:            lAxisX = (LONG)value - 128;
        break;

    case 0x31:            lAxisY = (LONG)value - 128;
        break;

    case 0x32:         lAxisZ = (LONG)value - 128;
        break;

    case 0x35:         lAxisRz = (LONG)value - 128;
        break;

    case 0x39:            lHat = value;
        break;
    }
}
```

HidP_GetUsageValue works the same way HidP_GetUsages does. It extracts the usage in UsageMin and UsageMax and the current value of the HID control from the HID input report. For my little RumblePad, I have examined only UsageMin to distinguish between the axes.

The rest of the sample code that I did not explain draws the values extracted from the HID input report in an appropriate manner. That's it!

Further Considerations

I showed in this article that it is possible to use the Raw Input API for joystick input with a little help from the HIDClass driver. I suppose that this API is just a layer on top of the HID user mode library and below DirectInput. In fact, it is also possible to bypass the Raw Input API and do all HID communication by using only the HID user library. Just take a look at the HClient sample of the WDK. It does the same things I do to interpret HID input reports without using Raw Input. You can't get any lower-level interface when developing your game input system as the HID user mode library is the only interface that separates your application from kernel mode, although I doubt that using the HID user library directly will gain much speed.

Last but not least, I provide you with a small outline of the things you have to do when using the HID user mode library directly:

1. Enumerate the devices of the HIDClass Device Setup Class using the SetupAPI.
2. Get the instance path of a specific device via SetupDiGetDeviceInterfaceDetail.
3. Open a handle to the device via CreateFile using the instance path.

4. Use `ReadFile`, `WriteFile` and the `HidP_*` functions to communicate with the HID.
5. Close the handle.

Note that you cannot use this method on mouse or keyboard devices, because Windows uses these devices exclusively (see Top-Level Collections Opened by Windows for System Use).

References

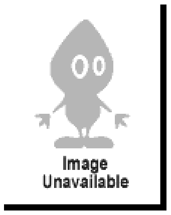
- Raw Input API at MSDN
- Human Input Devices at MSDN
- HID Usage Tables

History

- 23rd April, 2011: Initial post

License

About the Author



Alexander Böcken
Student
Germany

No Biography provided

Article Top

Like 0
Tweet 0
0

Sign Up to vote *Poor* ○ ○ ○ ○ ○ *Excellent* Vote

Comments and Discussions

Hint: For improved responsiveness ensure Javascript is enabled and choose 'Normal' from the Layout dropdown and hit 'Update'.
You must Sign In to use this message board.

Search this forum Go

☑ Profile popups Spacing Relaxed Noise Medium Layout Normal Per page 25 Update

First Prev Next

How about force feedback?	**_Almaz_**	**12-Jul-12 22:38**
I've been looking for a way to unify polled and event driven devices...	**roberts1964**	**6-May-12 6:42**
Link to HID Usage Tables Has Changed	**brettaur**	**29-Mar-12 7:20**
So many Errors	**XTREME104**	**1-Mar-12 3:55**
Re: So many Errors	XTREME104	1-Mar-12 3:57
Great article, but something isn't right.	**ProgramMax**	**23-Apr-11 12:13**
Re: Great article, but something isn't right.	ProgramMax	23-Apr-11 12:59
Re: Great article, but something isn't right.	ProgramMax	23-Apr-11 14:09

Last Visit: 31-Dec-99 18:00   Last Update: 17-Jun-13 13:41   Refresh   1

General   News   Suggestion   Question   Bug   Answer   Joke   Rant   Admin