Development of a Windows Device Driver for the Nintendo Wii Remote

Entwicklung eines Windows Treibers für die Nintendo Wii Remote

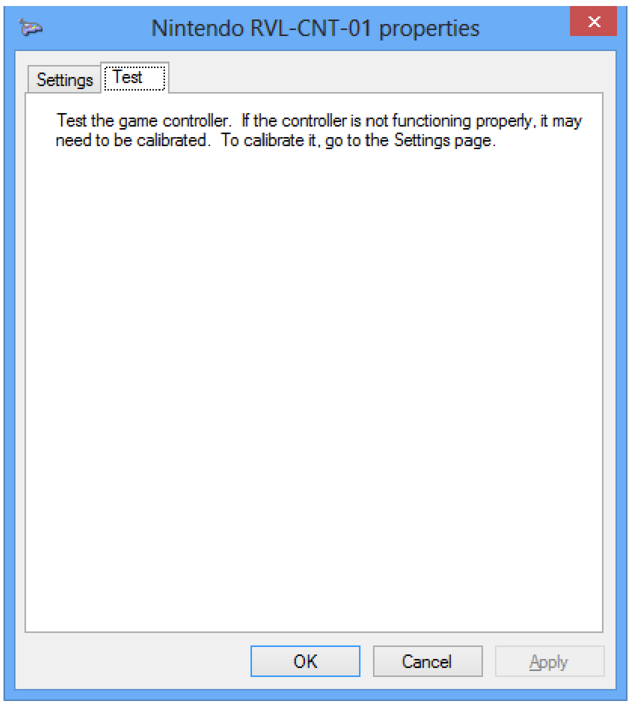
Julian Löhr
School of Informatics
SRH Hochschule Heidelberg
Heidelberg, Germany
julian.loehr@outlook.com

***Abstract*—This paper is about the development of a device driver for the Nintendo Wii Remote on Windows PC's.**

Keywords*—*Windows driver development, Wii Remote, human interface device, game controller, Bluetooth

I. INTRODUCTION

Many PC games do support game controllers. The Nintendo Wii Remote is a wireless controller for the Nintendo Wii console and the Nintendo Wii U console. It features several buttons, acceleration sensors and an infrared sensor. Furthermore it is possible to expand the controller via an additional port with various attachments. Those attachments are, i.e. the Nunchuk, a controller with additional buttons and acceleration sensors, a Classic Pro Controller, which is similar to an Xbox 360 controller, and other expansions. Thus it seems to be an alternative to other game controllers for controlling pc games.



Test the game controller. If the controller is not functioning properly, it may need to be calibrated. To calibrate it, go to the Settings page.

Fig. 1 Test mask of the preferences dialog for the Wii Remote with default driver

The Wii Remote uses Bluetooth for its wireless communication and is thereby connectable with a pc[1]. Windows does recognize the Wii Remote as a game controller, but as shown in Fig. 1 no inputs are exposed. Therefore it is not usable without any third party support. There are various programs to enable the Wii Remote to be used within video games, but all of them just map the inputs to keyboard keys[2]. So this is useful for some single-player games, but does not support analog input[2]. Additionally if multiple controllers are needed, e.g. for local multiplayer games like FIFA, this solution is not sufficient enough[2].

So the objective is to develop a device driver to enable it as a native game controller. Its input shall be exposed for games and multiple Wii Remote shall be distinguished.

II. PROBLEM

The Wii Remote communication is based on the Human Interface Device (HID) protocol[1]. HID is a device class for USB and Profile for Bluetooth and is used for input and output devices operated by humans[3]. It is self-describing, meaning every device tells the host via descriptors about his inputs and outputs[3]. Therefore every device can have different layouts but is operated by the same driver.

As in Table I, the Wii Remote violates this standard in various ways. The major violation is the incomplete report descriptor[1]. So Windows does know it is a game controller, but has no information about its layout. Further violations are that only the interrupt channel is used[1]. The interrupt channel is designed to only carry input and output reports[4]. But the Wii Remote uses some output reports for calibration[1], whereas feature reports should be used for that[4]. Those feature reports in turn must be send via the control channel[4]. Additionally the Wii Remote uses an output report to request an input report[1]. For that kind of action the standard state a GET_REPORT Message[4]. At last when either an expansion is connected or removed, the Wii Remote waits for a specific output report[1]. Until that report is received the device won't send any input reports anymore[1]. Such behavior is not designated by HID. Those violations render the default HID driver inoperative.

TABLE I. WII REMOTE HID VIOLATIONS

Wii Remote Behavior	**HID Standard Violation**
Report Descriptor states only sizes but no content	Report Descriptor incomplete
Output Reports used for calibration	Feature Reports should be used for that; Feature Reports must be sent via the Control Channel
Only uses Interrupt Channel	Interrupt Channel is designed to only carry Input and Output Reports
Output Report used to request an Input Report	GET_REPORT Message should be used
On connecting or removing expansions, an specific Output Report is awaited.	Such behavior is not part of HID

III. FINDING A SOLUTION

To enable the game controller functionality the driver shall intercept and correct the communication somewhere in the driver stack so it conforms HID.

A. *Driver Stack Basics*

At first the current driver stack of the Wii Remote was observed to find possible driver locations. A driver stack consists of multiple device nodes. Each device node represents a hardware device or one of its features, meaning a hardware device can be represented by multiple device nodes. As in Fig. 1 those device nodes are at least composed of a Function Driver Object (FDO) and a Physical Device Object (PDO)[5].

The FDO is the data representation of the function driver, which in turn is the main driver for that device node. The PDO is FDO of the underlying device node, establishing a connection between parent and child nodes in the stack[6].

Each function driver can be extended by a minidriver. Minidrivers shall handle device specific behavior and features, whereas the Function Driver might be a generic driver[7].

Between the FDO and PDO a filter driver can be placed. Filter drivers are meant to intercept and alter the communication between those device nodes[5].

B. Wii *Remote Driver Stack*

In Fig. 2 the default driver stack for the Wii Remote is shown.

At the bottom is the Bluetooth Enumerator node. It is the top end of the Bluetooth stack, representing the Bluetooth bus, which enumerates connected devices.

In the middle is the Bluetooth HID-Device, which is representing the Bluetooth Profile of the connected device, i.e. the Wii Remote. Its function driver is the HID Class driver with the Bluetooth minidriver. The HIDClass.sys driver implements the generic HID behavior, whereas HidBth.sys utilizes the Bluetooth specific communication.

On top is the HID-conformal game controller device. It is the exposed game controller part of the Wii Remote. This device node has no observable function driver. It was assumed that Windows retrieves the input information from that node.

C. *The Attempts*

Four attempts were devised based on the observable default driver stack for the Wii Remote.

1) *First Attempt*

The first attempt was to use a User Mode Driver Framework (UMDF) HID minidriver for the HID-conformal game controller, see number 1 in Fig. 2. This attempt failed because all read attempts were answered by STATUS_NOT_SUPPORTED.

2) *Second Attempt*

The second one was like the first one but to use an UMDF HID filter driver instead, see number 1 in Fig. 2. This failed as well as passing by the request resulted in a status code STATUS_NO_SUCH_PRIVILEGE.

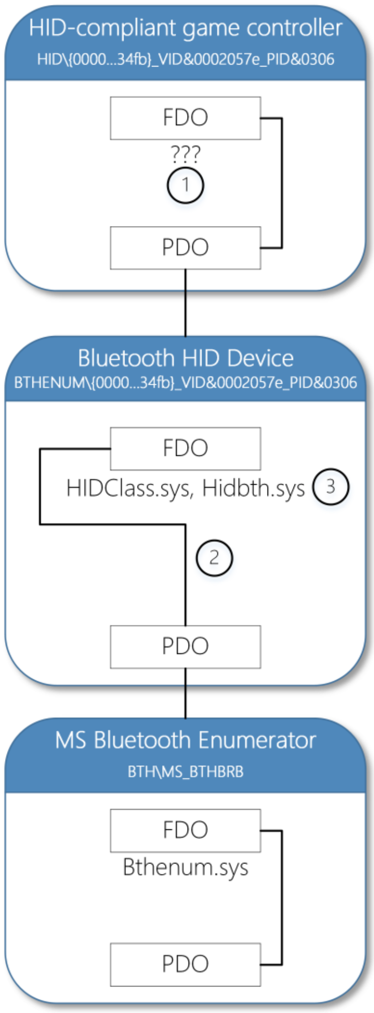


Fig. 2: Observable default driver stack for the Wii Remote

3) *Third Attempt*

Another attempt was to use a Kernel Mode Driver Framework (KMDF) filter driver between the HID Class driver and the Bluetooth bus driver (Bthenum.sys), see number 2 in Fig. 2. But it was not possible to load the filter driver and the default HID Class driver, at the same time.

4) *Fourth Attempt*

The last attempt was to replace the HidBth minidriver, see number 3 in Fig. 2, with a self-implemented KMDF minidriver. It was possible to successfully open a communication channel to the Wii Remote and exchange reports. Furthermore passing report descriptors and reports to the HID Class driver succeeded as well. Therefore this attempt was chosen as an appropriate solution.

IV. Draft

The conceived driver consists of three layers, seen in Fig. 3. The lowest layer abstracts the Bluetooth communication. In the middle is a layer, which handles all Wii remote specific things like parsing Wii Remote reports and responding with appropriate ones. The top layer is in charge of processing requests from the HID Class driver. Aside from that is a fourth part called Device. Its purpose is to handle all communication with the Plug and Play Manager and Power Management. Those Layers are described in the following sections.

A. *Bluetooth Layer*

At the bottom is the Bluetooth layer. It implements the Bluetooth communication, i.e. opening and closing the communication channel and sending and receiving reports from and to the Wii Remote.

To communicate with the Bluetooth device a Logical Link Control and Adaption (L2CAP) Channel is used. It is opened on startup and then a continuous reader is started. A continuous reader is an endless loop of sending out a read request. These read requests are blocking, meaning a request will only be completed in case of an error or when new data is available. In the case of an error the Wii Remote is considered as gone. The loop will terminate and the Device section shall be notified to signal the Plug and Play (PnP) Manager that the device was removed. The problem is that the PnP Manager won't recognize elsewhere the Wii Remote is gone, i.e. powered off, batteries dead or out the Bluetooth range. The only hint is that the L2CAP Channel is closed and the read request will return with an error. Apart from that new data is forwarded to the Wiimote layer to be processed. After processing, the read request is send out again to receive the next data.

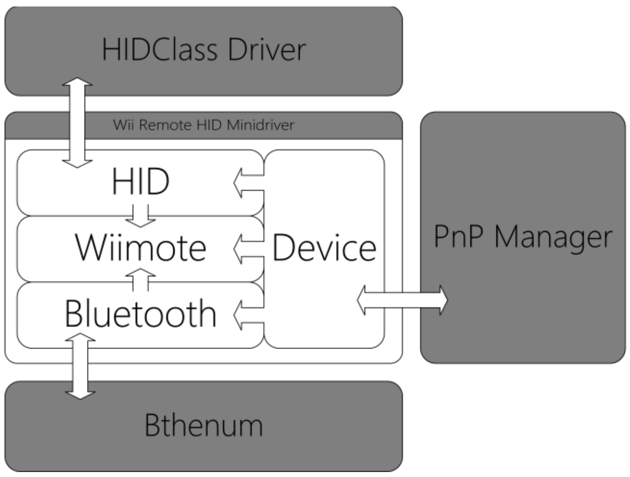


Fig. 3: Layout of the driver with its layers

B. *Wiimote Layer*

The middle layer is Wiimote. It handles all Wii Remote specific features and caches the current state. The layer is receiving input reports from the Bluetooth layer and processes them.

Wii Remote data reports (ID 0x30 – 0x3f) are containing input data. When receiving one of those the cached state is updated and the HID layer is notified about the input changes.

Status information (ID 0x20) contains generic information about the current state, i.e. battery level and the kind of connected extension. It is send by the Wii Remote on request or when the extension changes. First of all the battery level is updated. The four LEDs on the Wii Remote shall represent the current battery level. For that purpose a battery timer shall be implemented. It requests a status information message every minute to check if the battery level changed. On receiving a status information message the timer is reset. If the battery level is very low, i.e. only one LED left, the timer is stopped. After processing the battery level, it is checked whether the extension has changed. If so the report mode is set and the possible extension is initialized.

C. *HID Layer*

The top layer is called HID and it processes incoming requests from HID Class. It is composed of a default I/O queue, a read request buffer queue and a flag signaling if a new read request is processed immediately.

The default I/O queue receives any new requests. Only the internal device control request packages are processed, because those will contain I/O Control Codes (IOCTL), which in turn are used to get the report descriptor and the input reports. All other requests are processed by the framework. The important IOCTL is IOCTL_HID_READ_REPORT, which requests a new input report from the device. HID Class uses interrupt requests to retrieve changes in the input, meaning it sends out a read request to retrieve a new input report. When it is answered, it will send out another read report request immediately. To fulfill such behavior the read request buffer queue is used to buffer read request. Only when the Wiimote layer signals it has changed it state, the read request will be processed and completed. Processing the read request means parsing the current Wiimote state and creating an appropriate input report for HID Class. In case the Wiimote state changes when there is no read request currently buffered, a flag is used to signal whether a newly received read request shall be processed and completed immediately. Else an input change would be delayed until the input changes again, resulting in the delay or drop of the first input.

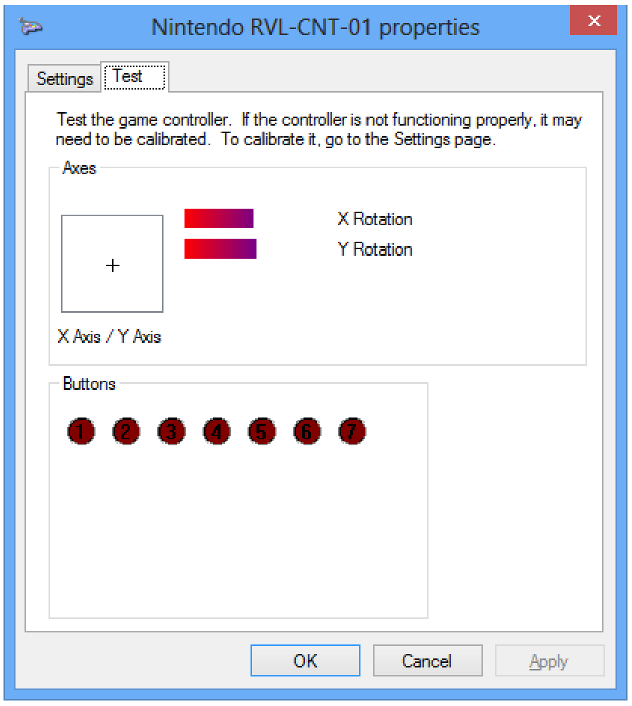
D. Device

The Device part communicates with the PnP Manager and the Power Management. It is the entry point for a newly connected device, creating and initializing all other layers. Moreover it will forward any events and changes of the device status to the layers. Evaluation

A basic implementation was made following the draft, featuring the eleven core buttons. The Implementation was made with the Windows Driver Kit 8 and Version 1.11 of the Windows Driver Framework. The driver was tested on Windows 7 and 8. For the test the input test mask of the game controller preferences dialog, as shown in Fig. 4, was used. All input was signaled without delay and in time. There were no errors while testing. So even the driver was not fully implemented, it proves the draft.

Because the driver only exposes the Wii Remote to the DirectInput API, it only works for older games. Modern games are using the XInput API, which is giving access to Xbox 360 Controllers. So it is not possible to control modern games with this driver.

Another flaw is the lack of driver signing. A code signing certificate cost money. Without being signed, the usage of a driver is difficult, because Windows won't load it. To use it anyway, the driver signing validation has to be turned off every time the driver will be used. However this can be resolved by acquiring such code signing certificate.


Nintendo RVL-CNT-01 properties

Settings | Test

Test the game controller. If the controller is not functioning properly, it may need to be calibrated. To calibrate it, go to the Settings page.

Axes

X Rotation
Y Rotation
X Axis / Y Axis

Buttons

Fig. 4: Test mask of the preferences dialog of the Wii Remote with the implemented driver installed

V. Outlook

A proper und full implementation of the driver combined with the driver being signed would be a proper alternative for older game controllers and capable to be used to control pc games. It would have the capability to be used by a broad user base.

One idea, which emerged through this paper, is to develop another draft or refactor the current draft, so the Wii Remote is recognized as an Xbox 360 controller. This may require reverse engineering the XUSB Driver.

Another idea is to use this draft for the Wii U Gamepad or Wii U Pro Controller. Especially the Wii U Pro Controller is very similar to the Xbox 360 controller.

References

[1] WiiBrew. Wiimote [Online]. Availabe: http://wiibrew.org/wiki/Wiimote [Accesed: 29.09.2013 23:06]

[2] J Löhr, „Wii Remote am Windows PC: Analyse und Vergleich verschiedener Programme und Libraries," unpublished.

[3] *Universal Serial Bus (USB): Device Class Definition for Human Interface Devices (HID)*, Version 1.11, 2001.

[4] *HUMAN INTERFACE DEVICE PROFILE 1.1*, Revision V11, 2012.

Microsoft Corporation. Device nodes and device stacks [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/ff554721(v=vs.85).aspxf [Accessed: 02.10.2013 11:48].

[5] Microsoft Corporation. Driver stacks [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/hh439632(v=vs.85).aspx [Accessed: 02.10.2013 12:01].

[6] Microsoft Corporation. Minidrivers, Miniport drivers, and driver pairs [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/hh439643(v=vs.85).aspx [Accessed: 02.10.2013 13:15].