



Le génie pour l'industrie

# **Reverse Engineering and Design of a Linux Driver for the PS4 USB Dongle for Guitar Hero Live**

**Presented to**

Prof. Pascal GIARD

**Prepared by**

**Daniel NGUYEN**

*daniel.nguyen.1@ens.etsmtl.ca*

École de technologie supérieure

Montréal, September 30, 2021

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Rhythm Games	4
1.2	Guitar Hero	4
1.3	Clone Hero and Hardware	4
1.4	Current State and Objective	5
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	About the PS3 & Wii U dongles	6
2.2	Sniffing USB Traffic	6
2.3	Tools	7
2.3.1	GreatFET One and FaceDancer	7
2.3.2	Wireshark and usbmon	7
2.3.3	Beagle USB 480 and Data Center	8
<b>3</b>	<b>First Attempt : Sniffing the USB Traffic Using the GreatFET One</b>	<b>9</b>
3.1	Procedure, Obstacles and Solutions	9
3.1.1	FaceDancer	9
3.1.2	A 20-Second Window	9
3.1.3	Data Analysis with ViewSB	10
3.1.4	Unexpected Data Results	10
3.2	Results	11
<b>4</b>	<b>Second Attempt : Sniffing the USB Traffic Using the Beagle USB 480</b>	<b>12</b>
4.1	Comparison Between GreatFET One and Beagle USB 480	12
4.1.1	Hardware Setup	12
4.1.2	Run Time	12
4.1.3	Data	13
4.2	Leads and Follow-up	13
4.2.1	Status Difference	13
4.2.2	Authentication Repetition	14
4.2.3	Mapping the HID Reports	14
4.3	“Magic” Data	14
<b>5</b>	<b>Adding PS4 Support</b>	<b>16</b>
5.1	Changes for DKMS Initial Release	16
5.1.1	hid-ghlive/src/hid-ids.h	16
5.1.2	hid-ghlive/src/hid-ghlive.c	16
5.2	Changes to DKMS in Preparation for Linux Kernel Inclusion	18
5.2.1	Implementation of Dynamic Endpoint Retrieval	18
5.2.2	Complete Code Overhaul	20
5.2.3	Linux Kernel Tree Patch	21
<b>6</b>	<b>Conclusion and Recommendations</b>	<b>23</b>

## A Full authentication sequence

26

# Summary

Since its release in 2005, the Guitar Hero (GH) franchise grew to become popular rhythm games on gaming consoles for children and adults alike. In 2015, Guitar Hero Live (GHL) was released, but sales failed to meet expectations and the franchise faded away from consoles in the following years. PC-based equivalents, notably Clone Hero (CH) released in 2017, gave players and enthusiasts a platform to play on using the console hardware and a community to play with. However, over time, hardware became scarce due to wear and tear. Moreover, driver compatibility issues limited the hardware that functioned on PC. This project aims to create a Linux based driver for the PlayStation (PS) 4 GHL USB dongle. The successful completion of this project will enable those dongles to be used instead of collecting dust which is important for the CH community and the promotion of circular economy.

The PS4 dongles are nearly functional. Every button and axis movement can be registered by the PC when activated individually. However, when fret buttons are pressed, the strum bar no longer registers. This combination is crucial to play the game as it mimics the strumming of notes and is the problem to be solved. It is known that the guitar controllers over the different consoles are identical and only the USB dongles differ.

The driver for PS3/Wii U dongles already exists. A “magic” packet is sent periodically to the dongle via a control USB request block (URB) which enables the dongle to function correctly, in other words, for the combination of frets and strum to work. The assumption is that the PS4 driver should function similarly, but with a different “magic” packet.

To retrieve the “magic” packet sent by the PS4, a device is connected as a man-in-the-middle (MITM) between the PS4 and USB dongle. The data sent between the two are intercepted and analyzed to identify the “magic” packet. Two different devices were used as a MITM due to technical difficulties with the first. After in depth analysis of the intercepted data, the “magic” packet was found being sent as an interrupt URB instead of a control URB. With the “magic” packet identified, the Dynamic Kernel Module Support (DKMS) driver was modified to add support for the PS4 dongle and released on May 21, 2021. This release allowed users with the PS4 dongle to use their hardware and play CH.

While preparing the code for Linux kernel inclusion, an issue was found with regards to memory allocation and a complete overhaul was undertaken in order to fix the issue and simplify the code. Once the fix patch was accepted, it was possible to add the PS4 support to the Linux kernel and conclude the project.

This report takes an in-depth look at the steps taken to complete this project and puts emphasis on the obstacles encountered as well as the solutions used to overcome them. With detailed explanations of the thought process used, the hope is that this report may one day help others that are attempting a similar project and serve as a learning tool.

# Chapter 1

## Introduction

The goal of this project is to create a Linux based driver for the PlayStation (PS) 4 USB dongle for Guitar Hero Live (GHL). To reach this goal, data sniffing is used to figure out what the dongle needs to behave correctly. The term sniffing refers to the process of monitoring and capturing all data packets that are passing through a computer network using packet sniffers [1]. Then, updates are made to the drivers in two steps : the Dynamic Kernel Module Support (DKMS) and the official Linux kernel. In this chapter, an introduction into rhythm games, Guitar Hero (GH), Clone Hero (CH), and the current state of affairs will be presented.

### 1.1 Rhythm Games

Rhythm games have existed since the 1970s, were very popular in Japan and made their way to North America. The first rhythm games were based on dancing and research has found that dancing games increase energy expenditure over that of traditional video games, and that they burn more calories than walking on a treadmill [2]. Guitar games have been used alongside physical therapy to help recovering stroke patients because of the multiple limb coordination required [3]. Furthermore, these games often inspire people to learn to play guitar and studies have shown that playing an instrument may be one of the best ways to help keep the brain healthy [4]. GH is the most well-known guitar game and will be discussed in detail as it is the basis of the project.

### 1.2 Guitar Hero

In 2005, GH was released in North America and was an instant success. The guitar controllers had an in-line 5-button configuration. Many sequels and band-specific installments were released throughout the following years and were available on the major gaming consoles : Sony, Microsoft, and Nintendo. In 2009, a decline in sales caused the industry of rhythm games to take a hiatus until 2015 when GHL was released for PS3, PS4, Wii U, Xbox 360, Xbox One, and Apple's iOS platforms. GHL revived the industry with a new 6-button guitar configuration (3 in series by 2 parallel) which created a whole new play style. The game did not sell as well as predicted and the release of Rock Band (RB) 4 around the same time didn't help. Both game publishers blamed each other for the poor sales performances. In December 2018, Activision, the GHL publisher, shut down their servers, which reduced the available song list from almost 500 to 42 present on disc, essentially killing the franchise. The GH community turned to CH, a PC-based clone of Guitar Hero. CH is the game for which this project aims to indirectly contribute to and will be discussed in the following section.

### 1.3 Clone Hero and Hardware

CH was released in 2017 as a replacement for Frets On Fire (FoF) and Frets On Fire X (FoFiX) which supported 4-lane drums. CH gained popularity due to its ability to play community-made songs like its predecessors. Currently, the CH Discord server has over 155 thousand members. Although the game is PC-based and can be played with a keyboard, most instruments from GH and RB are supported. Therefore, the demand for hardware remains high, but

hardware is becoming scarce due to wear over time. The demand has caused the prices of guitar controllers to rise and certain controllers are even becoming collectible items. The most recent 6-button guitar controllers that were released with GHL also function with CH. It is important to note that the limiting factor is not the guitar itself, but the required USB dongle. The guitar controllers are the same across all platforms, but the dongles are different for each. Currently, the Xbox 360, PS3 and Wii U dongles have drivers to support them. The PS4 dongles are the most common dongles available and require support. Successful completion of the project will allow users who own those dongles to put them to good use. The following section will discuss how things were at the start of this project and the primary objective.

## 1.4 Current State and Objective

The PS4 dongles nearly work. In fact, when the dongle is connected to a PC, actions to the buttons (frets) and the strum bar can be detected and their states read individually. However, when frets are held down, the strum bar no longer reports its state. In other words, the combination of frets and strum does not function, which makes gameplay impossible. The problem is that the dongle is not running at its full potential and limiting inputs. The assumption is that the PS4 dongle works similarly to the PS3/Wii U dongle where it requires a periodic “magic” message in order to behave correctly. The main objective of this project is to create a Linux based driver for the PS4 USB dongle to function properly. In order to achieve the main objective, the project is divided into 4 secondary objectives :

- O1 : Sniff the data between the console and the dongle
- O2 : Analyze the data to understand how the dongle works
- O3 : Implement the code required to support PS4 dongles in the DKMS driver
- O4 : Submit a patch for Linux kernel inclusion

### Outline

In the remainder of this report, [chapter 2](#) provides the necessary background information required to understand the project. It presents the methodology and tools used throughout. Next, [chapter 3](#) describes the steps taken to capture data using the GreatFET One [\[5\]](#) device in an attempt to achieve **O1** and **O2**. It describes the obstacles and limitations found using that device. [Chapter 4](#) pursues the steps using the Beagle USB 480 Protocol Analyzer [\[6\]](#) to obtain key data thus completing **O1** and **O2**. It compares the differences between the two devices and presents the “magic” data. Finally, [chapter 5](#) covers the modifications brought to the driver code for initial release to accomplish **O3** as well as inclusion into the official Linux kernel to complete **O4**. It explains the code in greater detail as well as the discussions had with Linux kernel maintainers.

## Chapter 2

# Background

This chapter presents the background information that is required to understand the project. It explains how the driver for the other dongles works, establishes the method for sniffing USB traffic between the console and the PS4 dongle, and presents the hardware and software tools used in [chapter 3](#).

### 2.1 About the PS3 & Wii U dongles

Initially, the Microsoft Xbox 360 dongles and their 6-button guitar controllers from GHL were the only ones to work out of the box. Since all the guitar controllers are identical, people with controllers from PlayStation or Nintendo had to purchase an Xbox dongle and sync their controller to the new dongle in order to play CH. In 2020, Professor Pascal Giard created a driver for the GHL USB dongles of the PS3 and Wii U platforms to be used on the Linux platform. The driver can be found on Github [7]. During the creation of the driver, Professor Giard found that the USB dongles for Wii U and PS3 were identical. This was also independently found by Emma, a.k.a. InvoxPlayGames, who is the main author and maintainer of the GHLtarUtility repository. GHLtarUtility is a program that emulates an Xbox 360 controller while using another on the Windows platform. In order for the dongle to fully function, the driver sends a control packet with a specific 8 bytes of data and value in the form of a USB request block (URB) every 10 seconds. This “magic” data and “magic” value were found using RPCS3 (a PS3 emulator), Wireshark and usbmon (a Linux kernel module). Wireshark and usbmon will be introduced in detail in the following sections. With a better understanding on how the PS3/Wii U dongles work, the next section presents the method to sniff USB traffic between the PS4 console and the PS4 dongle.

### 2.2 Sniffing USB Traffic

Based on the driver created for PS3 and Wii U, the assumption is that the PS4 dongle (device) works in a similar fashion. The PS4 runs on a closed operating system (OS). The OS is proprietary to Sony and access to the drivers built within is restricted. Therefore, the proposed method is to sniff out the data sent between the host and the device in order to locate the data packet that enables full operation and replicate it through a driver. In order to accomplish this, the GreatFET One [5] is setup as a man-in-the-middle (MITM) between the PS4 and the dongle. Then, the FaceDancer [8] software that comes with the GreatFET One is used in order to intercept the data and analyze it. [Figure 2.1](#) presents the physical hardware setup used.

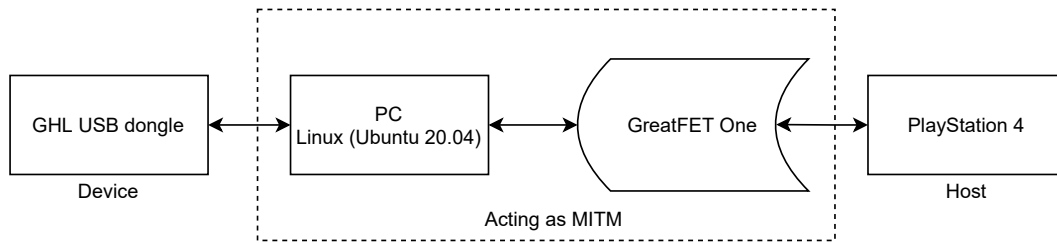


Figure 2.1: GreatFET One hardware setup adapted from FaceDancer README [8]

Before any test is done, the PlayStation (PS) 4 console, USB dongle, and guitar controller were tested in order to ensure functionality. Armed with an understanding of how the data is going to be sniffed, the next section describes the tools used.

## 2.3 Tools

### 2.3.1 GreatFET One and FaceDancer

The GreatFET One, presented in [fig. 2.2](#), is a device created by Great Scott Gadgets (GSG). It can be used as an interface to an external chip, a logic analyzer, or a debugger. It is capable of supporting Hi-Speed USB which makes it the right tool for this project because the USB dongle runs on Full-Speed which is slower. A case was 3D printed in order to protect the GreatFET One. The stereolithography (STL) files can be found on Thingiverse [9].

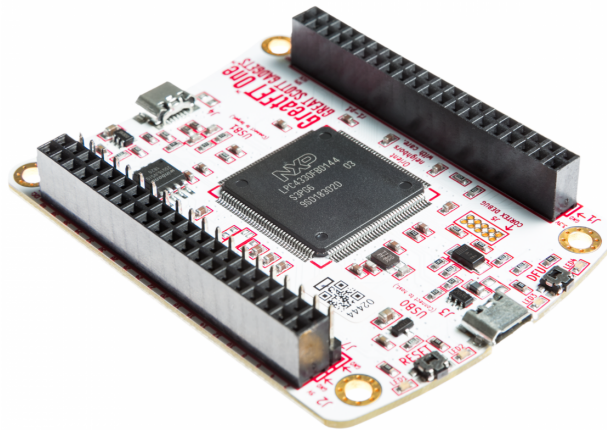


Figure 2.2: GreatFET One

FaceDancer is the software that enables the GreatFET One to perform tasks such as emulation and USB proxying. The most recent version of FaceDancer can be found on GitHub [8] in the usb-tools repository maintained by the team from GSG. The software is written in Python and it is known to lack documentation although some examples are provided. The USB proxy feature is noted as being complete.

### 2.3.2 Wireshark and usbmon

Wireshark is a widely-used network protocol analyzer, but has been capable of capturing USB traffic on Linux since version 1.2.0 using the Linux usbmon interface. This software neatly arranges the data into blocks and allows filters to be applied to keep or hide specific blocks. usbmon is the Linux kernel module used to collect traces of input and output on the USB bus. Using the pair allows to view USB traffic in a user-friendly way.



### 2.3.3 Beagle USB 480 and Data Center

The Beagle USB 480 Protocol Analyzer is made by Total Phase [6] and is the hardware capable of capturing and interactively displaying Hi-Speed USB bus states and traffic in real time. The Data Center [10] software is a graphical user interface (GUI) for the Beagle analyzer. It parses and displays the captured data in blocks that are managed and easier to analyze. The software is proprietary and does not interact with other hardware devices. In fact, the software must detect a compatible device to connect with in order to function. It therefore can only see data coming from the Beagle. Furthermore, the Beagle repackages the data into bulk URBs before sending it to the analysis PC where Data Center software unpacks the data. Lastly, the file type is different than what is captured using Wireshark and usbmon. This means that Data Center cannot parse data captured through Wireshark and vice versa.

## Chapter 3

# First Attempt : Sniffing the USB Traffic Using the GreatFET One

In this chapter, the first attempt to sniff the USB traffic in order to detect the “magic” data using the GreatFET One [5] and to achieve objective **O1** and **O2** is discussed. In the first section, the procedure to sniff data, the obstacles, and the solutions to the obstacles are presented. The second section explains the results and conclusions drawn which ultimately lead to the purchase of another protocol analyzer which will be discussed in [chapter 4](#).

### 3.1 Procedure, Obstacles and Solutions

In this section, the procedure to begin sniffing, the encountered obstacles, and the possible solutions that justify the steps followed are discussed. Each subsection presents a particular obstacle and the solution found to conquer it. With the physical setup [fig. 2.1](#) connected and all necessary dependencies installed, the first attempt to run FaceDancer to enable a USB proxy is performed and fails.

#### 3.1.1 FaceDancer

The first obstacle encountered was getting the FaceDancer software to function. In order to run in USB Proxy mode, it is necessary for the user to specify the vendor ID (VID) and the product ID (PID) of the USB device to proxy. This information can be retrieved using the command `lsusb` in the Linux Terminal with the dongle inserted. The VID is 0x1430 (RedOctane) and the PID is 0x07bb. The `facedancer-usbproxy.py` python script needs root access in order to execute correctly. However, it is also necessary to maintain the environment, thus the `sudo -E ./facedancer-usbproxy.py -v 0x1430 -p 0x07bb` is used. This allowed the dongle (device) to begin communicating with the PS4 (host) and live data streamed through the terminal. Once the guitar controller is activated, data packets can be seen and the host recognizes the controller as being connected. The live-streaming data representation does not give a clear indication of the information being transferred between the host and the device. The sheer amount of data flowing in hexadecimal format is overwhelming. Although data can be witnessed, the sample time is limited. The following subsection discusses the 20-second window of data and what seems to be causing it.

#### 3.1.2 A 20-Second Window

It is quickly noticed that once the guitar controller is activated, there is a limited amount of time before the controller is no longer responsive. In fact, every test done shows approximately 20 seconds of active function before the controller becomes unresponsive. This behaviour means that the GreatFET One is not being completely transparent throughout the proxy. In other words, the data that flows between the PS4 and the USB dongle is getting modified when it passes through the GreatFET One. When the USB dongle is directly connected to the PS4, the guitar controller remains responsive at all times. With the GreatFET One acting as a MITM, the loss of responsiveness proves that the data is not passing through completely which is preventing normal operation. Other tests are performed in order to verify the USB proxy mode. A failed attempt to proxy a USB Mass Storage device to a laptop as well as another failed attempt to

proxy a wired controller to the same laptop proves that there is an issue with either the software or the hardware. Either the FaceDancer software is modifying and therefore corrupting the data or there is a physical hardware problem with the GreatFET One meaning that it is defective. Upon further research, many open issues relevant to USB proxy mode were found on the FaceDancer GitHub [11]. The conclusion is that either the software is not functioning properly, or the hardware is defective. Data can still be collected, but it is possible that the important data needed to accomplish the project isn't being sniffed. An attempt is made to reach out to the creators and maintainers of the GreatFET One and FaceDancer tools through their Discord server [12] to get support, but without success. It is found that the team has moved on to another project and support for the GreatFET One was not available. In fact, the original creator of USB Proxy was no longer with the company. Considering this information and wanting to be efficient, a decision is made to acquire a more mature device capable of performing the man-in-the-middle (MITM) task. A test was performed on another GreatFET One device to rule out the possibility that the hardware is simply defective. The second device acted the exact same way as the first which consequently leads us to believe that there is more likely a problem with the software than two defective devices. Considering that during the 20-second window, the guitar controller functions as it should, it is believed that the "magic" data is present somewhere and can be retrieved. The next subsection discusses data analysis using ViewSB.

### 3.1.3 Data Analysis with ViewSB

As mentioned previously, the live streaming data seen through the terminal is not a convenient representation and is difficult to analyze as is. It is possible to write a parser that can simplify the representation by bundling the data into understandable blocks and filter them thereafter. Writing such a parser for this project was not necessary because it already exists. ViewSB [13] is a tool provided by the team at GSG which was created to perform such a task. After many attempts to use ViewSB with FaceDancer to view the data and after asking for help on the GSG Discord server, Mikaela "Qyriad" Szekely, a software engineer at GSG, confirmed that ViewSB's USBProxy backend is broken at the moment and would not be usable. Considering this information, it becomes necessary to use a different parsing tool : Wireshark and usbmon.

### 3.1.4 Unexpected Data Results

Upon analysis of the data captured using Wireshark and usbmon and considering that we expect to find a periodic control packet with "magic" data, it appears that there are no control packets being repeated with static data. In fact, in the 20-second interval of time that is sniffable, the SET report data is always different and no pattern can easily be detected. Figure 3.1 shows an example of the report sequence that is witnessed.

35 445.029549	host	3.2.0	USBHID	64 GET_REPORT Request
36 445.030277	3.2.0	host	USBHID	112 GET_REPORT Response
2387 452.367946	host	3.2.0	USBHID	64 GET_REPORT Request
2388 452.368378	3.2.0	host	USBHID	72 GET_REPORT Response
2707 453.375549	host	3.2.0	USBHID	128 SET_REPORT Request
2708 453.376035	3.2.0	host	USBHID	64 SET_REPORT Response
3023 454.380699	host	3.2.0	USBHID	128 SET_REPORT Request
3024 454.381213	3.2.0	host	USBHID	64 SET_REPORT Response
3339 455.388883	host	3.2.0	USBHID	128 SET_REPORT Request
3340 455.389353	3.2.0	host	USBHID	64 SET_REPORT Response
3657 456.394622	host	3.2.0	USBHID	128 SET_REPORT Request
3658 456.395119	3.2.0	host	USBHID	64 SET_REPORT Response
3973 457.398722	host	3.2.0	USBHID	128 SET_REPORT Request
3974 457.399209	3.2.0	host	USBHID	64 SET_REPORT Response
4287 458.404858	host	3.2.0	USBHID	128 SET_REPORT Request
4288 458.405384	3.2.0	host	USBHID	64 SET_REPORT Response
4605 459.410201	host	3.2.0	USBHID	128 SET_REPORT Request
4606 459.410694	3.2.0	host	USBHID	64 SET_REPORT Response
4923 460.423824	host	3.2.0	USBHID	128 SET_REPORT Request
4924 460.424349	3.2.0	host	USBHID	64 SET_REPORT Response
5555 462.427802	host	3.2.0	USBHID	64 GET_REPORT Request
5556 462.428204	3.2.0	host	USBHID	80 GET_REPORT Response
6183 464.431060	host	3.2.0	USBHID	64 GET_REPORT Request
6184 464.432011	3.2.0	host	USBHID	128 GET_REPORT Response

Figure 3.1: Report sequence captured between PS4 and USB dongle using Wireshark

In an attempt to spot a pattern, five runs were performed and compared visually. The SET reports began the same way and appeared to have a counter in them and many bytes left empty. The first and second GET reports returned the same response from the device and the third GET report would sometimes return the same 64 byte response. Upon further research involving the DualShock 4 (DS4), it was found that Sony protects their system from using cloned and aftermarket controllers [14].

The PS4 issues a challenge to the controller and awaits its response in order to authenticate it. If the controller fails the challenge, it gets blocked after a certain amount of time. Therefore, the 20-second window that is experienced through USB Proxy appears to be the allocated time for the controller to be authenticated or not.

The hypothesis is that the first GET report request is the PS4 letting the controller know to get ready for a challenge. The controller responds with a ready message or something in that order. The 8 following SET reports is the challenge. The SET reports are numbered 0 to 7 because the order is important. The controller responds with a handshake every time it receives data. Using cyclic redundancy check (CRC), the device must complete the calculations, proving its authenticity. The second GET report request is the PS4 asking the controller if it's ready to answer the challenge, and the response is always the same return message. The final GET report request is the host asking the device for the challenge answer. It appears to be at this moment that the connection is severed, meaning that the authentication failed.

Research was done by others to understand the authentication process for a DS4 [14]. They concluded that the PS4 issues 5 SET reports as the challenge and expects 19 GET reports in return. This information doesn't exactly match what is witnessed, but does confirm that it is the authentication process. The authentication process is not important for this project because the hardware is authentic, but simply not functioning correctly. Therefore, the data does not seem to hold the key to success. The next section discusses the results drawn from the tests and proposes an alternative.

## 3.2 Results

Seeing as the guitar controller becomes unresponsive during the MITM, it is concluded that the GreatFET One is not fully transparent. In view of this conclusion and considering the open issues on GitHub, it is logical to suppose that the data seen using this hardware may not be complete. Another device is ordered to verify the data sniffed by the GreatFET One : the Beagle USB 480 Protocol Analyzer [6]. The next chapter presents the continuation of the sniffing process using the Beagle USB 480.

## Chapter 4

# Second Attempt : Sniffing the USB Traffic Using the Beagle USB 480

In this chapter, the second attempt to sniff the USB traffic using the Beagle USB 480 [6] and to achieve objective **O1** and **O2** is discussed. The first section compares the results between the two analyzers. The second section presents the leads found and the follow-up performed. Finally, the last section presents the “magic” data which allows us to add the PS4 support to the DKMS driver which will be seen in [chapter 5](#).

### 4.1 Comparison Between GreatFET One and Beagle USB 480

In this section, a comparison is made between the GreatFET One [5] and Beagle USB 480 in terms of the hardware setup, the duration of a run and the data observed.

#### 4.1.1 Hardware Setup

The Beagle USB 480 is setup in a similar fashion to the GreatFET One, but with the difference that the GHL USB dongle is plugged directly into the device and not through the PC used for analysis. [Figure 4.1](#) below represents the wired connections.

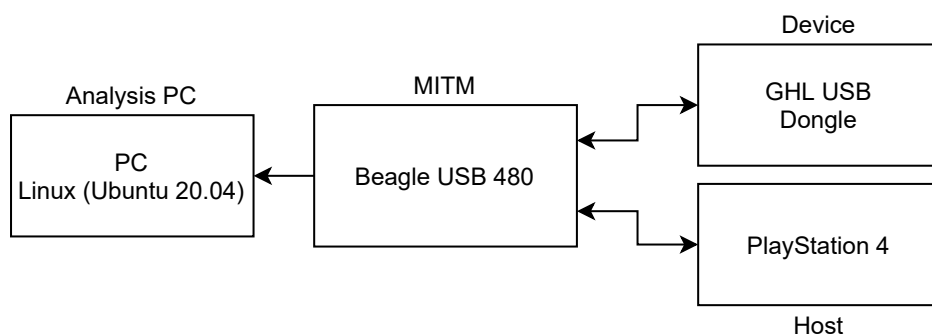


Figure 4.1: Beagle USB 480 Hardware Setup

#### 4.1.2 Run Time

It is possible to play indefinitely while capturing data. The guitar controller doesn't become unresponsive which ensures that the data observed using the device is complete and can be trusted. This is an improvement compared to

the limited 20-second window provided by the GreatFET One. A data comparison can now be performed to either validate or discredit the GreatFET One observed data which will be seen in following subsection.

### 4.1.3 Data

When comparing the control URBs captured by the GreatFET One and by the Beagle USB 480, it is observed that there is no new data. In fact, the sequence and results are exactly the same with only one visible difference : the response to GET STATUS. [Figure 4.2](#) and [fig. 4.3](#) show the difference in status.

00	Get Configuration Descriptor	Index=0 Length=9
00	Get Configuration Descriptor	Index=0 Length=41
00	Get Device Status	
00	SETUP txn	80 00 00 00 00 00 02 00
00	IN txn [6 POLL]	01 01
00	OUT txn	
00	Set Configuration	Configuration=1

Figure 4.2: Beagle GET STATUS response

26	4.779275	3.9.0	host	USB	105 GET_DESCRIPTOR Response
27	4.782421	3.9.0	host	USB	64 GET STATUS Request
28	4.782581	3.9.0	host	USB	66 GET STATUS Response
Frame 28: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on :					
USB URB					
wStatus: 0x0001					

Figure 4.3: GreatFET One GET STATUS response

When using the Beagle, the returned status is 0x0101 whereas the returned status while running the GreatFET One or during replay attempts is 0x0001. It is possible that the status difference is key.

With the ability to capture longer samples of data, the authentication process can be observed in its entirety. [Figure A.1](#) shows the full authentication process occurrences over a long sample.

It is observed that the authentication sequence occurs every 75 seconds and the challenge answer is returned in 33 separate packets contrary to the single packet observed through the GreatFET One.

The difference in status and the repetitive authentication are the leads found through observing the control URB and are pursued in more detail in the following section.

## 4.2 Leads and Follow-up

In this section, the leads found and their follow-ups are discussed. Each subsection presents a particular lead and the process followed to pursue it.

### 4.2.1 Status Difference

As seen previously, the response to the GET STATUS packet defers when using the Beagle USB 480 and the GreatFET One. It was logical to suppose that the status has an affect on the function of the dongle. Therefore, a replay was done in an attempt to obtain the wStatus = 0x0101 in userspace using libusb. The attempt was unsuccessful and the status remained 0x0001. Research was done in order to better understand what the status meant. In fact, the byte that differs represents the remote wakeup feature. The hypothesis is that the feature should not have an effect on solving our problem and the choice to not pursue this lead any further is made. The authentication process is the next lead to be checked.

## 4.2.2 Authentication Repetition

Knowing that the authentication repeats itself every 75 seconds, the assumption is that it is this process that allows the dongle to function properly. A replay was done which repeats the authentication process with the 8 packets representing the challenge and the 33 return packets representing the answer. The guitar could still not perform the combination of fret and strum which confirms that the authentication process is not the issue. This was expected because even with the GreatFET One, before the authentication could be completed, the dongle was functioning correctly.

With dead ends on both leads, it was decided to approach the problem in a different way. Instead of observing the control URBs, the plan is to verify the mapping of the human interface device (HID) reports. The HID reports, which represent the state of the buttons and axes on the guitar controller, are sent to the host in the form of a 64-byte interrupt URBs. The assumption is that the mapping is different with the dongle and that the HID generic driver in charge of mapping the device is missing the information. the following subsection discusses the process of mapping the HID reports.

## 4.2.3 Mapping the HID Reports

In order to map the HID reports, it is necessary to start with a baseline where no buttons are pressed and the axes are stable. From there, 1 button is pressed at a time and the reports are analyzed to find out which byte is changed in order to map the bit that is changed. While performing the mapping, the interrupt URBs are not filtered away because they are of interest. Once each individual button and axis is mapped, the combination of fret and strum is next. While observing the reports in order to identify the combination, output reports are seen. Figure 4.4 below shows the filtered output reports.





















Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
0	0:00.000.000					 Capture started (Aggregate)	[Wed 19 May 2021 02:22:00 PM]
600	0:02.533.303	9 B		08	02	 Output Report [48]	
1228	0:03.869.469	9 B		08	02	 Output Report [48]	
1229	0:03.869.469	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00
3711	0:09.142.126	9 B		08	02	 Output Report [48]	
3712	0:09.142.126	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00
7546	0:17.151.124	9 B		08	02	 Output Report [48]	
7547	0:17.151.124	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00
11385	0:25.160.123	9 B		08	02	 Output Report [48]	
11386	0:25.160.123	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00
15213	0:33.169.121	9 B		08	02	 Output Report [48]	
15214	0:33.169.121	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00
19071	0:41.178.120	9 B		08	02	 Output Report [48]	
19072	0:41.178.120	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00
22936	0:49.187.119	9 B		08	02	 Output Report [48]	
22937	0:49.187.119	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00
26670	0:57.196.119	9 B		08	02	 Output Report [48]	
26671	0:57.196.119	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00
30372	1:05.205.119	9 B		08	02	 Output Report [48]	
30373	1:05.205.119	9 B		08	02	 OUT txn	30 02 08 0A 00 00 00 00 00

Figure 4.4: Output reports filtered

This interrupt packet is sent from the host to the device and contains a 9-byte data message. The output report also repeats itself every 8 seconds with the same data message and could potentially be the “magic” data that allows the dongle to function. The next section discusses the test performed with the data to achieve objective O2.

## 4.3 “Magic” Data

The assumption being that the output report found in the previous section is the “magic” data that unlocks the dongle, a test is created in user space that sends the interrupt packet to the dongle using libusb functions. Right after the code is executed, the combination of fret and strum is tested successfully and functions as intended for a few seconds. The success of the user space test confirms that the data found in the output report is the “magic” data. The difference

between what we were expecting and reality is that instead of the “magic” data being sent through a control URB, it is actually sent via an interrupt URB.

Considering this information, another look at the initial data retrieved from the GreatFET One was performed. The “magic” data was found as an interrupt output and was filtered away due to the initial assumption that the “magic” data traveled on a control URB. Over the 20-second window provided by the GreatFET One, the “magic” data is seen on 4 frames out of over 7000 frames which made it nearly impossible to find while scrolling. In theory, the project could’ve been completed without the Beagle USB 480.

With the “magic” data found and validated, modifications can be made to the DKMS driver. The details of the changes made in the driver code will be detailed in [chapter 5](#).



## Chapter 5

# Adding PS4 Support

In this chapter, the changes made to support the PS4 dongle in the DKMS driver for initial release, the changes made in anticipation to Linux kernel inclusion, and the final changes are presented.

It is important to note that there are slight differences between the initial release code and the final code because the DKMS driver needed to be released quickly in order to have the CH community validate its functionality before being included into the Linux kernel. Therefore, certain variables such as the endpoint address and bInterval value were hard-coded instead of retrieved. Details about these variables will be discussed in the following sections.

### 5.1 Changes for DKMS Initial Release

The changes made for the DKMS driver release to achieve objective **O3** are discussed in this section and are divided by file name. The figures present in this section are taken from the comparison of the GitHub GH Live DKMS code [7] before and after the added PS4 support. It is important to note that not all changes are presented in this report. Trivial changes such as modifications to comments, version number, copyright information, rules, and module description were made, but not documented here. To see all the changes made, visit the GitHub project [7] commit on May 21, 2021. Moreover, a potential problem with the stable kernels  $\geq 5.11$  was detected by Professor Giard when he updated the DKMS to reflect his previous kernel inclusion. This problem was isolated as coming from certain memory allocations and causing a hard freeze. Further investigation into the problem will be done when preparing for Linux kernel inclusion in [section 5.2](#) and resolved in the next version of 5.13.

#### 5.1.1 hid-ghlive/src/hid-ids.h

In the header file, the VID and PID were defined for the PS4 dongle. [Figure 5.1](#) shows the additional lines.

```
@@ -4,4 +4,7 @@
#define USB_VENDOR_ID_SONY_GHLIVE          0x12ba
#define USB_DEVICE_ID_SONY_PS3WIIU_GHLIVE_DONGLE 0x074b

{+#define USB_VENDOR_ID_REDOCTANE_GHLIVE          0x1430+}
{+#define USB_DEVICE_ID_REDOCTANE_PS4_GHLIVE_DONGLE 0x07bb+}

#endif
```

Figure 5.1: Addition of VID and PID

Note that the vendor ID (VID) is listed as RedOctane and not Sony.

#### 5.1.2 hid-ghlive/src/hid-ghlive.c

In the .c file, a bit is assigned to the PS4 dongle and the poke interval is reduced to 8 seconds. [Figure 5.2](#) shows the changes made.

```
#define {+GHL_GUITAR_PS4 BIT(3)+}
{+#define+} GHL_GUITAR_PS3WIIU BIT(2)
#define GHL_GUITAR_CONTROLLER BIT(1)

#define GHL_GUITAR_POKE_INTERVAL [-10-]{+8+} /* In seconds */
```

Figure 5.2: Addition of bit and reduction of poke interval

Initially, the poke interval was set to 10 seconds. The data showed that the output report reoccurs every 8 seconds. Tests were performed at 10, 9, and 8 second intervals in order to verify if the poke needed to be shortened. At 10-second intervals, we could witness loss of fret and strum functionality every time. At 9-second intervals, we could witness loss of fret and strum functionality from time to time. At 8-second intervals, no loss of data was found. Any loss is unacceptable and therefore 8 seconds was chosen as the value. This shortened interval affects the PS3/Wii U dongles as well, but was considered acceptable in order to simplify the code instead of having 2 poke intervals.

The “magic” data was added as an array. Figure 5.3 presents the 9-byte “magic” data retrieved.

```
@@ -31,6 +33,11 @@
static const char ghl_ps3wiiu_magic_data[] = {
    0x02, 0x08, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00
};

{+/* PS4 magic data found through usb sniffing */+}
{+static const char ghl_ps4_magic_data[] = {+}
{+    0x30, 0x02, 0x08, 0x0A, 0x00, 0x00, 0x00, 0x00, 0x00+}
{+};+}
```

Figure 5.3: Addition of “magic” data array

Contrary to the PS3/Wii U driver which sends the “magic” data via control URB, the PS4 must send an interrupt URB. An interrupt URB does not have a setup packet, but instead has information such as the endpoint address and bInterval value for that endpoint. Therefore, in order to simplify the code and not allocate memory for a setup packet uselessly, the decision was made to have distinct poke and callback functions for the PS3/Wii U and PS4 dongles. So, the original poke and callbacks are renamed with the added “ps3wiiu” tag and new functions are created for the PS4 support. Figure 5.4 and fig. 5.5 present the major changes between the PS3/Wii U and PS4 poke functions : the pipe and the fill function.

```
{+    unsigned int pipe = usb_sndintpipe(usbdev, 0x02);+}
```

Figure 5.4: Declaration of pipe using usb\_sndintpipe()

When using the usb\_sndintpipe() function, the endpoint address is the second specified parameter. For the DKMS driver, the address is hard-coded as 0x02 which represents endpoint 2 as an OUT.

```
{+    usb_fill_int_urb(+}
{+        urb, usbdev, pipe,+}
{+        sc->databuf, poke_size,+}
{+        ghl_magic_poke_cb_ps4, NULL, 5); /* the bInterval for the EndPoint is 5. */+}
{+    ret = usb_submit_urb(urb, GFP_ATOMIC);+}
```

Figure 5.5: Interrupt URB fill function

The usb\_fill\_int\_urb() is similar to the usb\_fill\_control\_urb() except that the interrupt version does not require the setup packet and instead requires a bInterval value. For endpoint 2, the bInterval = 5 and is found using lsusb. The address and bInterval values will need to be retrieved instead of hard-coded for Linux kernel inclusion and will be discussed in section 5.2. Since the interrupt URB does not require a setup packet, the callback function only frees the transfer buffer.

In the probe function, the `mod_timer()` remains common and it is here that we differentiate between the PS3/Wii U and PS4 dongles using the quirks and an `if...else if` condition. Figure 5.6 shows the added condition.

```
@@ -162,11 +217,13 @@
static int ghlive_probe(struct hid_device *hdev,
    }

    if (sc->quirks & GHL_GUITAR_PS3WIIU) {
        timer_setup(&sc->poke_timer, [-ghl_magic_poke,-]{+ghl_magic_poke_ps3wiiu,+} 0); [-mod_timer(&sc->poke_timer,-)
[-      jiffies + GHL_GUITAR_POKE_INTERVAL*HZ);-]
    }
    {+else if (sc->quirks & GHL_GUITAR_PS4){+}
    {+      timer_setup(&sc->poke_timer, ghl_magic_poke_ps4, 0);+}
    {+      } +}
    {+      mod_timer(&sc->poke_timer,+
    {+          jiffies + GHL_GUITAR_POKE_INTERVAL*HZ);+}
    {+          return ret;
    }
}
```

Figure 5.6: Addition of condition to verify dongle type

The quirks represent the `driver_data` which are defined in the `hid_device_id` structure. Figure 5.7 presents the addition of the PS4 device to the structure.

```
static const struct hid_device_id ghlive_devices[] = {
    { HID_USB_DEVICE(USB_VENDOR_ID_SONY_GHLIVE, USB_DEVICE_ID_SONY_PS3WIIU_GHLIVE_DONGLE),
      .driver_data = GHL_GUITAR_CONTROLLER | GHL_GUITAR_PS3WIIU},
    {+{ HID_USB_DEVICE(USB_VENDOR_ID_REDOCTANE_GHLIVE, USB_DEVICE_ID_REDOCTANE_PS4_GHLIVE_DONGLE),+}
    {+      .driver_data = GHL_GUITAR_CONTROLLER | GHL_GUITAR_PS4 },+}
    { }
};
```

Figure 5.7: Addition of PS4 device

This concludes the changes required to have a functional DKMS driver. The next section presents the changes required for Linux kernel inclusion, notably the retrieval of the endpoint address and `bInterval` value discussed previously.

## 5.2 Changes to DKMS in Preparation for Linux Kernel Inclusion

The changes made to the DKMS driver in preparation for Linux kernel inclusion are discussed in this section and are based on the additions to the DKMS driver seen in section 5.1. The first subsection discusses the changes made to replace the hard-coded endpoint address and `bInterval` values with a dynamic method. The second subsection presents the complete overhaul of the code to fix a detected hard freeze. Finally, the third section describes the two patches submitted for kernel inclusion.

### 5.2.1 Implementation of Dynamic Endpoint Retrieval

Using `xpad.c` and `hid-corsair.c` as examples, the OUT endpoint was retrieved dynamically instead of being hard-coded. First, the `ghlive_sc` structure received a new data item. Figure 5.8 represents the addition of an endpoint descriptor structure.

```

@@ -44,6 +44,7 @@ struct ghlive_sc {
    int device_id;
    struct timer_list poke_timer;
    struct usb_ctrlrequest *cr;
    {+struct usb_endpoint_descriptor *ep_irq_out;+}
    u8 *databuf;
};

```

Figure 5.8: Modified ghlive\_sc structure

The `struct usb_endpoint_descriptor` is added in order to save the information about the endpoint which will be retrieved during the probe function and used in the poke function.

In the probe function, the newly added `ep_irq_out` is initialized to `NULL`. Then, the following lines are added in the `else if` condition. Figure 5.9 presents the block of code that dynamically retrieves the endpoint from the USB interface.

```

else if (sc->quirks & GHL_GUITAR_PS4){

    {+struct usb_interface *intf = to_usb_interface(sc->hdev->dev.parent);+}

    {+
        if (intf->cur_altsetting->desc.bNumEndpoints != 2){+}
        return -ENODEV;+}
    {+
        +}
    {+
        for (i = 0; i < intf->cur_altsetting->desc.bNumEndpoints; i++) { +}
        {+
            struct usb_endpoint_descriptor *ep =+}
            {+
                &intf->cur_altsetting->endpoint[i].desc;+}
        {+
            +}
        {+
            if (usb_endpoint_xfer_int(ep)) { +}
            {+
                if (usb_endpoint_dir_out(ep)){+}
                {+
                    sc->ep_irq_out = ep;+}
                }+}
        {+
            }+}
        timer_setup(&sc->poke_timer, ghl_magic_poke_ps4, 0);
    }
}

```

Figure 5.9: Retrieving endpoint descriptor

A `usb_interface` structure is declared and retrieved using the `to_usb_interface()` function. The interface allows the driver to verify the number of endpoints and to validate that there are in fact 2 endpoints. Next, a for loop iterates through the available endpoints and finds the OUT endpoint, which gets saved to the `ep_irq_out`. With the OUT endpoint descriptor in memory, it is possible to replace the hard-coded values `0x02` and `5` to their dynamically retrieved values. Figure 5.10 presents the replacement of the hard-coded values.

```

@@ -123,7 +124,7 @@ static void ghl_magic_poke_ps4(struct timer_list *t)
    struct usb_device *usbdev = to_usb_device(sc->hdev->dev.parent->parent);
    const u16 poke_size =
        ARRAY_SIZE(ghl_ps4_magic_data);
    unsigned int pipe = usb_sndintpipe(usbdev, [-0x02;-]{+sc->ep_irq_out->bEndpointAddress;+})

    sc->databuf = kzalloc(poke_size, GFP_ATOMIC);
    if (!sc->databuf) {
@@ -142,7 +143,7 @@ static void ghl_magic_poke_ps4(struct timer_list *t)
    usb_fill_int_urb(
        urb, usbdev, pipe,
        sc->databuf, poke_size,
        ghl_magic_poke_cb_ps4, NULL, [-5]; /* the bInterval for the EndPoint is 5. */-){+sc->ep_irq_out->bInterval;+}
    ret = usb_submit_urb(urb, GFP_ATOMIC);
}

```

Figure 5.10: Replacing values

The next step is to investigate the possibility of a hard freeze situation in the kernel tree.

## 5.2.2 Complete Code Overhaul

As mentioned in [section 5.1](#), a potential problem causing a hard freeze was detected. A clone of the kernel tree repository was compiled and the problem confirmed. The memory allocations made in the `ghl_magic_poke()` function were causing the hard freeze. After discussion, the choice was made to completely overhaul the code by simplifying memory allocation and usage. Instead of allocating memory to build a URB periodically in the `ghl_magic_poke()` function, the URB is built once during the `ghlprobe_probe()` function and re-used periodically. Furthermore, the allocations will be done using the `devm_kzalloc()` function in order for the device to manage the memory. There is no need to free memory allocated with that function. The primary changes to `hid-ghlive.c` are shown here. For a complete view of all the changes, visit the GitHub project [\[7\]](#) commit on June 3, 2021.

To implement the simplification, it is necessary for the `ghlprobe_sc` structure to change. [Figure 5.11](#) presents the changes made.

```
struct ghlprobe_sc {
    struct hid_device *hdev;
    unsigned long quirks;
    [-int device_id;-] {+struct urb *urb;+}
    struct timer_list poke_timer;
    [-struct usb_ctrlrequest *cr;-]
    [-struct usb_endpoint_descriptor *ep_irq_out;-]
    [-u8 *databuf;-]
};
```

Figure 5.11: Overhauling the `ghlprobe_sc` structure

The structure must contain the URB. The `poke_timer` is kept in order to maintain periodicity and the `hdev` and `quirks` are kept as well. The `device_id` is found to be an artefact from previous tests and is removed. The `cr`, `ep_irq_out`, and `databuf` are removed because they will be created during the probe function and will no longer be needed once the URB is built. This simplification also removes the need to have separate poke functions and callback functions for PS3/Wii U and PS4. [Figure 5.12](#) and [fig. 5.13](#) present the changes made to unify the functions.

```
static void [-ghl_magic_poke_ps3wiiu(struct-] {+ghl_magic_poke(struct+} timer_list *t)
{
    int ret;
    [-struct urb *urb;-]
    struct ghlprobe_sc *sc = from_timer(sc, t, poke_timer);
    [-struct usb_device *usbdev = to_usb_device(sc->hdev->dev.parent->parent);-]
    [-const u16 poke_size =-]
    [-    ARRAY_SIZE(ghl_ps3wiiu_magic_data);-]
    [-unsigned int pipe = usb_sndctrlpipe(usbdev, 0);-]

    [-sc->cr = kzalloc(sizeof(*sc->cr), GFP_ATOMIC);-]
    [-if (!sc->cr)-]
    [-    goto resched;-]

    [-sc->databuf = kzalloc(poke_size, GFP_ATOMIC);-]
    [-if (!sc->databuf) {-]
    [-    kfree(sc->cr);-]
    [-    goto resched;-]
    [-}-]

    [-urb-] {+ret+} = [-usb_alloc_urb(0,-] {+usb_submit_urb(sc->urb,+} GFP_ATOMIC);
    if [-(!urb) {-]
    [-    kfree(sc->databuf);-]
    [-    kfree(sc->cr);-]
    [-    goto resched;-] {+(ret < 0)+}
    {+
        hid_err(sc->hdev, "usb_submit_urb failed: %d", ret);+}
}
```

Figure 5.12: Unification of the pokefunction

The poke function only submits the URB and the callback function verifies the transfer status and reschedules the timer.

```
static void [-ghl_magic_poke_cb_ps3wiiu(struct-]{+ghl_magic_poke_cb(struct+} urb *urb)
{
    [-if (urb) {-]
        /* Free sc->cr and sc->databuf allocated in ghl_magic_poke_ps3wiiu() */-]
        kfree(urb->setup_packet);-]
        kfree(urb->transfer_buffer);-]
    [-]
    [-]-]{+struct ghlive_sc *sc = urb->context;+}

[-static void ghl_magic_poke_cb_ps4(struct urb *urb)-]
[-{-] if [-(urb) {-]
    [-
        /* Free sc->databuf allocated in ghl_magic_poke_ps4() */-]
        kfree(urb->transfer_buffer);-]
    [-]
    [-]{+(urb->status < 0)+}
    [+
        hid_err(sc->hdev, "URB transfer failed : %d", urb->status);+}

    [+
        mod_timer(&sc->poke_timer, jiffies + GHL_GUITAR_POKE_INTERVAL*HZ);+}
}
```

Figure 5.13: Unification of the callback function

Since the URB types required for the PS3/Wii U and PS4 are different, separate functions (`ghl_init_urb_ps3wiiu` and `ghl_init_urb_ps4`) are created to initialize the correct URB depending on the dongle type. These functions reuse the same logic that used to be inside the poke functions. Also, the recently added dynamic endpoint address retrieval code is moved out of the probe function and into `ghl_init_urb_ps4`. The probe function simply calls the correct URB initialization function and sets up the timer. As mentioned, using the `devm_kzalloc` function releases us from needing to manage the memory freeing. The only memory that needs to be managed is the memory allocated for the URB which is only freed when the device is removed. [Figure 5.14](#) presents the line that frees the URB within the `ghlive_remove` function.

```
@@ -248,13 +223,14 @@
static void ghlive_remove(struct hid_device *hdev)
{
    struct ghlive_sc *sc = hid_get_drvdata(hdev);

    del_timer_sync(&sc->poke_timer);
    {+usb_free_urb(sc->urb);+}
    hid_hw_close(hdev);
    hid_hw_stop(hdev);
}
```

Figure 5.14: Freeing URB memory upon removal of the device

With the DKMS driver simplified and tested, the last step is to include the changes to the Linux kernel in order to achieve objective O4.

### 5.2.3 Linux Kernel Tree Patch

The kernel inclusion was done in two parts due to the detection of the hard freeze. The first patch repairs the problem and the second patch adds PS4 support. Simply put, the overhauled code from the DKMS driver was implemented into `hid-sony.c` to fix the hard freeze caused by the memory allocations without the added support. Once the fix was applied to the kernel [15], the PS4 support could be added and a second patch submitted. All patches are sent via e-mail and these e-mails are archived. The PS4 support patch mailing list archive can be found on [marc.info](http://marc.info) [16].

After the submission of the PS4 support patch, it was brought to our attention that our code contained a lot of direct USB calls which was not ideal because they are not covered by regression tests. It was suggested that we use certain functions, notably `hid_hw_raw_request()`, that were covered by regression tests and a patch was made available to

us for testing. However, this patch caused a hard freeze and the following error message : `BUG: scheduling while atomic`. We attempted to pinpoint the source of the hard freeze by investigating every function called within it, and we suspect that the cause is the `usb_control_msg()`. In fact, the description of `usb_control_msg()` stipulates to not use this function from within an interrupt context [17] which is our case. The description also suggests the use of `usb_submit_urb()` for asynchronous messages which is the function that we are using.

Furthermore, we learnt that certain devices were capable of handling controls sent through interrupt and control transfers. We tested if it were possible to send the “magic” data using a control transfer, and it does function. Therefore, it is possible to modify the code to send as control transfer in order to share more code paths with the PS3/WiiU-dongle support. Also, we learnt that the `wValue` parameter of the control transfer for the PS3/WiiU (`wValue = 0x201`), which we believed to be a “magic” value, is not “magic”. We tested other values and the dongle still functioned as long as the “magic” data remains the same. Therefore, we could remove the “magic” value and have it generated automatically in the same way that `hid_hw_raw_request()` creates the `wValue` parameter.

A second version of the patch was created and the changes were also made to reflect on the DKMS driver. [Figure 5.15](#) and [fig. 5.16](#) present the important changes made in version 2.

```
static int ghl_init_urb(struct sony_sc *sc, struct usb_device *usbdev{+,+}
{+
    const char ghl_magic_data[], u16 poke_size+})
{
    struct usb_ctrlrequest *cr;
[-    u16 poke_size;-]
    u8 *databuf;
    unsigned int pipe;
    {+u16 ghl_magic_value = (((HID_OUTPUT_REPORT + 1) << 8) | ghl_magic_data[0]);+}

[-    poke_size = ARRAY_SIZE(ghl_ps3wiiu_magic_data);-]
    pipe = usb_sndctrlpipe(usbdev, 0);

    cr = devm_kzalloc(&sc->hdev->dev, sizeof(*cr), GFP_ATOMIC);
@@ -663,10 +674,10 @@ static int ghl_init_urb(struct sony_sc *sc, struct usb_device *usbdev)
    cr->bRequestType =
        USB_RECIP_INTERFACE | USB_TYPE_CLASS | USB_DIR_OUT;
    cr->bRequest = USB_REQ_SET_CONFIGURATION;
    cr->wValue = cpu_to_le16([-ghl_ps3wiiu_magic_value-]{+ghl_magic_value+});
    cr->wIndex = 0;
    cr->wLength = cpu_to_le16(poke_size);
    memcpy(databuf, [-ghl_ps3wiiu_magic_data-]{+ghl_magic_data+}, poke_size);
```

Figure 5.15: Replacing the “magic” value

Since the `ghl_init_urb()` is to be shared for the PS3/WiiU and PS4, the function receives two new parameters: the “magic” data and the array size (`poke_size`). These parameters will be filled during the probe function depending on the dongle.

```
if (sc->quirks & {+(+)}GHL_GUITAR_PS3WIIU {+| GHL_GUITAR_PS4+}) {
    sc->ghl_urb = usb_alloc_urb(0, GFP_ATOMIC);
    if (!sc->ghl_urb)
        return -ENOMEM;

    {+if (sc->quirks & GHL_GUITAR_PS3WIIU)+}
        ret = ghl_init_urb(sc, usbdev{+, ghl_ps3wiiu_magic_data,+}
            ARRAY_SIZE(ghl_ps3wiiu_magic_data));+}
{+
    else if (sc->quirks & GHL_GUITAR_PS4)+}
        ret = ghl_init_urb(sc, usbdev, ghl_ps4_magic_data,+}
            ARRAY_SIZE(ghl_ps4_magic_data));+}
{+
{+
```

Figure 5.16: Calling the `ghl_init_urb` function within `probe()`

The patch version 2 is simpler and allows for code sharing. The changes made for the DKMS driver can be found on the GitHub page [7]. On August 20, 2021 the patch was accepted and applied to next.

## Chapter 6

# Conclusion and Recommendations

### Conclusion

In this report, we presented the steps performed to reverse engineer and design a Linux driver for the PS4 USB dongle for Guitar Hero Live which was unsupported until this point. We presented the steps performed to sniff data using the GreatFET One and the Beagle USB 480 and successfully found the “magic” data to unlocking full functionality. As a result, we have the means to implement support for PS4 dongles as a DKMS driver and directly into the Linux kernel. This added support enables the use of the PS4 hardware and promotes circular economy.

It could have been possible to complete this project without the purchase of the Beagle USB 480 Protocol Analyzer since the “magic” data could be found using the GreatFET One. However, because the guitar controller could not maintain a connection longer than 20 seconds, it was logical to doubt the data retrieved and to continue assuming that the “magic” data was transmitted via control transfer.

### Recommendations

1. **Explore why the GreatFET One is not fully transparent**

The first attempts at sniffing data were carried out using the GreatFET One device and we noticed that a lack of transparency from the hardware or software was causing the communication between host and device to be broken after 20 seconds. The cause should be found so that we can put the devices to good use.



# Acknowledgement

I gratefully acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). I would like to thank my mentor, Professor Pascal Giard, for allowing me this opportunity to step out of my comfort zone and for inspiring me to pursue my studies beyond my purview. Your support and belief in me were instrumental in the success of this project.

# Bibliography

- [1] EC-Council, *What are sniffing attacks and their types?* EC-Council Official Blog, Jun. 2020. [Online]. Available: <https://blog.eccouncil.org/what-are-sniffing-attacks-and-their-types/#:~:text=Sniffing%20is%20the%20process%20of> (visited on 06/03/2021).
- [2] M. Clinic, *Adding activity to video games fights obesity, study shows*, ScienceDaily, Jan. 2007. [Online]. Available: <https://www.sciencedaily.com/releases/2007/01/070104144703.htm> (visited on 05/14/2021).
- [3] G. Boston, *Healing with guitar hero*, The Washington Times, Sep. 2008. [Online]. Available: <https://www.washingtontimes.com/news/2008/sep/24/you-dont-have-to-be-a-quick-fingered-15-year-old-t/> (visited on 05/14/2021).
- [4] S. Sapega, *Playing an instrument: Better for your brain than just listening*, Penn Medicine News, Jan. 2017. [Online]. Available: <https://www.pennmedicine.org/news/news-blog/2017/january/playing-an-instrument-better-for-your-brain-than-just-listening> (visited on 05/14/2021).
- [5] *GreatFET One*, 2021. [Online]. Available: <https://greatscottgadgets.com/greatfet/one/>.
- [6] *Beagle USB 480 protocol analyzer*, 2021. [Online]. Available: <https://www.totalphase.com/products/beagle-usb480/>.
- [7] *HID GHLive DKMS*, 2021. [Online]. Available: <https://github.com/evilynux/hid-ghlive-dkms>.
- [8] *Facedancer*, 2021. [Online]. Available: <https://github.com/usb-tools/Facedancer>.
- [9] *Greatbox STL*, 2019. [Online]. Available: <https://www.thingiverse.com/thing:3835955>.
- [10] *Total phase data center*, 2021. [Online]. Available: <https://www.totalphase.com/products/data-center/>.
- [11] *Facedancer issues*, 2021. [Online]. Available: <https://github.com/usb-tools/Facedancer/issues>.
- [12] *Great scott gadgets discord server*, 2021. [Online]. Available: <https://discord.gg/rsfMw3rsU8>.
- [13] *ViewSB*, 2021. [Online]. Available: <https://github.com/usb-tools/ViewSB>.
- [14] *PS4 - controller authentication*, 2018. [Online]. Available: <https://forum.gimx.fr/viewtopic.php?f=3&t=2384&sid=56a525b470c20a9dd01f7843ffc49387>.
- [15] *Mailing list archive : Fix patch*, 2021. [Online]. Available: <https://marc.info/?t=162282312900003&r=1&w=2>.
- [16] *Mailing list archive : Ps4 support patch*, 2021. [Online]. Available: <https://marc.info/?t=162637933100002&r=1&w=2>.
- [17] *Usb\_control\_msg(9)*, 2021. [Online]. Available: [https://manpages.debian.org/jessie-backports/linux-manual-4.8/usb\\_control\\_msg.9.en.html](https://manpages.debian.org/jessie-backports/linux-manual-4.8/usb_control_msg.9.en.html).

## Appendix A

# Full authentication sequence

Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
2676	0:14.006.142	8 B		05	00	Get Feature Report [243]	
3158	0:15.007.259	64 B		05	00	Set Feature Report [240]	Length=64
3646	0:16.007.363	64 B		05	00	Set Feature Report [240]	Length=64
4127	0:17.007.486	64 B		05	00	Set Feature Report [240]	Length=64
4608	0:18.007.602	64 B		05	00	Set Feature Report [240]	Length=64
5080	0:19.007.726	64 B		05	00	Set Feature Report [240]	Length=64
5562	0:20.007.875	64 B		05	00	Set Feature Report [240]	Length=64
6050	0:21.007.971	64 B		05	00	Set Feature Report [240]	Length=64
6531	0:22.008.083	64 B		05	00	Set Feature Report [240]	Length=64
7475	0:24.008.340	16 B		05	00	Get Feature Report [242]	
8414	0:26.009.630	64 B		05	00	Get Feature Report [241]	
8896	0:27.010.702	64 B		05	00	Get Feature Report [241]	
9393	0:28.011.818	64 B		05	00	Get Feature Report [241]	
9876	0:29.012.920	64 B		05	00	Get Feature Report [241]	
10358	0:30.014.032	64 B		05	00	Get Feature Report [241]	
10844	0:31.015.157	64 B		05	00	Get Feature Report [241]	
11330	0:32.016.296	64 B		05	00	Get Feature Report [241]	
11805	0:33.017.449	64 B		05	00	Get Feature Report [241]	
12287	0:34.018.504	64 B		05	00	Get Feature Report [241]	
12762	0:35.019.660	64 B		05	00	Get Feature Report [241]	
13256	0:36.020.776	64 B		05	00	Get Feature Report [241]	
13739	0:37.021.908	64 B		05	00	Get Feature Report [241]	
14225	0:38.022.989	64 B		05	00	Get Feature Report [241]	
14707	0:39.024.144	64 B		05	00	Get Feature Report [241]	
15189	0:40.025.247	64 B		05	00	Get Feature Report [241]	
15671	0:41.026.416	64 B		05	00	Get Feature Report [241]	
16161	0:42.027.503	64 B		05	00	Get Feature Report [241]	
16638	0:43.028.615	64 B		05	00	Get Feature Report [241]	
17115	0:44.029.736	64 B		05	00	Get Feature Report [241]	
17598	0:45.030.857	64 B		05	00	Get Feature Report [241]	
18068	0:46.032.004	64 B		05	00	Get Feature Report [241]	
18524	0:47.033.090	64 B		05	00	Get Feature Report [241]	
19007	0:48.034.239	64 B		05	00	Get Feature Report [241]	
19484	0:49.035.361	64 B		05	00	Get Feature Report [241]	
19966	0:50.036.453	64 B		05	00	Get Feature Report [241]	
20443	0:51.037.573	64 B		05	00	Get Feature Report [241]	
20919	0:52.038.708	64 B		05	00	Get Feature Report [241]	
21402	0:53.039.810	64 B		05	00	Get Feature Report [241]	
21884	0:54.040.966	64 B		05	00	Get Feature Report [241]	
22359	0:55.042.063	64 B		05	00	Get Feature Report [241]	
22834	0:56.043.191	64 B		05	00	Get Feature Report [241]	
23316	0:57.044.291	64 B		05	00	Get Feature Report [241]	
23805	0:58.045.427	64 B		05	00	Get Feature Report [241]	
37701	1:28.056.146	8 B		05	00	Get Feature Report [243]	
38183	1:29.058.145	64 B		05	00	Set Feature Report [240]	Length=64
38660	1:30.059.244	64 B		05	00	Set Feature Report [240]	Length=64

Figure A.1: Full authentication sequence