Le génie pour l'industrie

Auto-Calibration Process for Mad Catz Fender Mustang Pro-Guitar

Presented to

Prof. Pascal GIARD

Prepared by

Daniel NGUYEN

daniel.nguyen.1@ens.etsmtl.ca

École de technologie supérieure

Montréal, September 30, 2021

Contents

Summary

As audio and video technology progressed, signal processing became more complex and time consuming. This processing introduces a delay that must be compensated through calibration in order to play rhythm games that mandate user interactions with high timing accuracy. As an attempt to automate the compensation process, the auto-calibration feature appeared in 2008 with the release of Rock Band (RB) 2 and was present on all later versions of the game. The feature simplified and refined the calibration process by having sensors integrated to the guitar controller. The sensors replaced the need for players to take actions during the process which removed human error caused by anticipation, stress to achieve accuracy and reaction time.

Rhythm games have seen a decline over the last few years and major developers are no longer releasing new titles. Despite this, the hardware has been given a new life thanks to games such as Clone Hero. Clone Hero is a PC-based rhythm game released in 2017 that supports various guitar controllers from different gaming consoles as well as drums. Interest has been shown by the main developer of Clone Hero to implement auto-calibration. However, the auto-calibration process is a black box and its inner workings are unknown. This project aims to understand how the auto-calibration process works.

To our knowledge, the details about the auto-calibration process are still uncharted and we are not aware of any other effort towards understanding its implementation. We therefore explored it in order to find out how it works from a software and hardware point of view. In order to study the auto-calibration, data was captured between the emulated console (host) and the dongle (device), then analyzed. The analysis allowed us to build a hypothesis and test it. A complete program was created to reproduce the process as proof of concept (PoC). Our PoC proves that we understand the auto-calibration process and are capable of re-creating it. This report will therefore describe our findings following our exploration. With detailed analysis of the data collected, it is possible to implement the auto-calibration feature and integrate the hardware into future works.

Chapter 1

Introduction

The goal of this project is to understand the auto-calibration process for Rock Band (RB) guitar controllers that support this feature in order to implement it into Clone Hero (CH). The understanding of the process will also allow makers of custom guitar controllers to implement the same functionality. To comprehend the auto-calibration process, the data exchanged between host and device is sniffed and analyzed. The term sniff refers to the process of monitoring and capturing all data packets that are passing through a computer network using packet sniffers [1]. The captured data allows us to deduce the sequence of events as well as the USB control messages required to perform the process.

This chapter explains the need for calibration, the hardware that is being calibrated, and the current state of affairs with regards to what is known about the calibration process.

1.1 About Calibration

In the context of this project, the goal of calibration is to compensate for audio and video latency for newer televisions and sound systems. As audio and video technologies progressed, signal processing became more complex and time consuming. This lag created due to signal processing is not usually a problem when watching television, but becomes an issue with video games. Modern televisions usually have a Game Mode which by-passes certain unessential video processors in order to limit input lag [2]. Game Mode, although not available for all TVs, does help reduce input lag, but for games that require a lot of precision, notably rhythm games, it is still greatly beneficial to calibrate audio and video.

The calibration process can be performed manually where a player must press a button upon a visual or auditory cue. This method introduces human error due to lack of consistency and precision. The auto-calibration feature was first released in 2008 with RB2. The guitar controller was capable of auto-calibrating using new hardware integrated inside. This new hardware is common to many following iterations of guitar controllers and most probably works in the same way.

The auto-calibration process is similar to the manual one, but without human error. In fact, it still uses auditory and visual cues, but instead of a player pressing a button as a response, the controller responds automatically. The auto-calibration consists of a two-step process. The first is the audio calibration where a chirp is generated every half-second by the game and is “heard” by the guitar controller through a microphone. Once the audio calibration is done, the video calibration follows, where the screen flashes white at regular intervals, and once again, the guitar controller “sees” the flashes through what appears to be a light sensor. The following section discusses hardware found on the guitar controllers.

1.2 Auto-Calibration Hardware

The RB2 guitar controllers were the first to implement auto-calibration. They were equipped with two sensors : a microphone and a light sensor. The microphone picked up the chirps during audio calibration and the light sensor detected the white flashes from the screen during video calibration. Figure 1.1 presents the location of the light sensor and the microphone.

Figure 1.1: Calibration hardware present on some RB guitars, where (1) the light sensor and (2) microphone are highlighted [3]

RB guitar controllers released after 2008 were all equipped with these sensors to allow auto-calibration. The sensors appear to be the same from RB2 through to RB4. For this project, the Mad Catz Fender Mustang Pro-Guitar is used and it is very likely that the process and data found will apply for other guitar controllers as well. However, this remains to be verified using different guitar controllers over the different platforms and is out of the scope of this project. The successful completion of this project will allow the auto-calibration feature to be implemented to Clone Hero and allow players with supported hardware to take advantage of the feature. Moreover, this project will enable creators of guitar controllers to integrate the same hardware to their designs and incorporate the auto-calibration feature as well. The following section discusses how things were at the beginning of this project and the main objective.

1.3 Current State and Objective

There is no information available with regards to the auto-calibration process. The process is a black box and needs to be discovered using data sniffing. The main objective of this project is to understand how the auto-calibration process works to enable an implementation of it in CH. In order to achieve the main objective, the project is divided into 3 secondary objectives:

- O1: Sniff the data between the console and the dongle during the auto-calibration process
- O2: Analyze the data to understand how the auto-calibration process works
- O3: Create a userspace program to mimic the auto-calibration process

Outline

In the remainder of this report, chapter 2 provides the necessary background information required to understand the project. It presents the tools used throughout and important information about the USB protocol. Next, chapter 3 describes the steps taken to achieve **O1** using the Beagle USB 480 Protocol Analyzer [4]. It explains the physical setup and emphasizes the obstacles encountered. Chapter 4 discusses the analysis of the data to accomplish **O2**. Specifically, it looks at the mapping and significance of the captured data with regards to the process to build hypotheses. Lastly, chapter 5 addresses the tests performed to verify the hypotheses through unit tests and a proof-of-concept userspace program to achieve **O3**. It explains the software implementation and the test results. Lastly, we close this report with concluding remarks and recommendations for future works in chapter 6.

Chapter 2

Background

In this chapter, the hardware and software tools, the emulated host, and the device are presented in detail. The tools used for data sniffing are the Total Phase Beagle USB 480 Protocol Analyzer and the Data Center software included with the analyzer. The emulated host is the Dolphin Emulator and the device is, as previously mentioned, the Mad Catz Fender Mustang Pro-guitar. The USB protocol, specifically the control and interrupt transfers, are also presented in order to better understand the analysis done in chapter 4.

2.1 Beagle USB 480 and Data Center Software

The Beagle USB 480 Protocol Analyzer [4] is made by Total Phase and is a device capable of capturing and interactively displaying Hi-Speed USB bus states and traffic in real time. The Data Center software [5] 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 cannot be used in conjunction with other hardware USB analyzers. 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. With the tools used for data sniffing presented, the next section discusses the host : Dolphin Emulator.

2.2 Dolphin Emulator

Dolphin [6] is a free and open-source video game console emulator for Nintendo GameCube and Wii that runs on Windows, Linux, MacOS, and Android. For this project, the emulated console is the Nintendo Wii. It could have been possible to use a real Nintendo Wii console instead of an emulated version, but in our case, it was simpler to emulate due to the challenges of working remotely. Auto-calibration is supported by all RB games except the first and any of those could have been used to observe the auto-calibration process. However, the guitar controller studied is compatible only with RB3. In fact, neither RB2 nor RB4 had charts for the pro-guitar. With a better understanding of the host, the following section presents the device : Fender Mustang Pro-Guitar.

2.3 Mad Catz Fender Mustang Pro-Guitar

The Pro-Guitar is right on the line between toy and instrument. Contrary to other previous guitar controllers that had 5 or 6 buttons to represent the frets, the Fender Mustang has 17 frets covering all 6 strings which are represented by 102 individual buttons on the neck. Moreover, instead of a simple strum bar, the Pro-Guitar has 6 strings that must be picked which makes it very similar to a real guitar. In fact, the guitar controller comes with a musical instrument digital interface (MIDI) out port which can be wired and used as a MIDI controller. Figure 2.1 presents the Pro-Guitar in a comparison with fig. 2.2, a standard RB3 guitar controller.

Figure 2.1: Mad Catz Fender Mustang Pro-Guitar

Figure 2.2: Rock Band 3 Standard Guitar

2.4 Communication with the Guitar Dongle

The wireless dongle communicates using the USB protocol. The USB protocol transfers messages known as USB request blocks (URBs). There are 4 types of URBs possible within the USB protocol : control, interrupt, isochronous and bulk. For this project, the control and interrupt types are the two used because the dongle is recognized as human interface device (HID) class. These two types also take into consideration the direction using IN and OUT, where IN types request data back from the device to the host and OUT types send data to the device from the host.

When a device is an HID class, the host periodically polls the device for its status in the form of an HID report sent as data through an IN interrupt transfer. The report varies in size depending on the device and its configuration. In our case, the HID report is 27 bytes long and gives information about all the buttons, switches, and sensors as will be shown in section 4.1.

The control transfers are used for command and status operations. In other words, it can be used to control the hardware. A control transfer is a multi-part message and requires a setup packet which details the type of request. Figure 2.3 illustrates the three transactions: setup, data, and status that build a control transfer. Each transaction contains three types of packets: token, data and handshake. The setup transaction's data packet is commonly called the setup packet and will be analyzed in section 4.4.

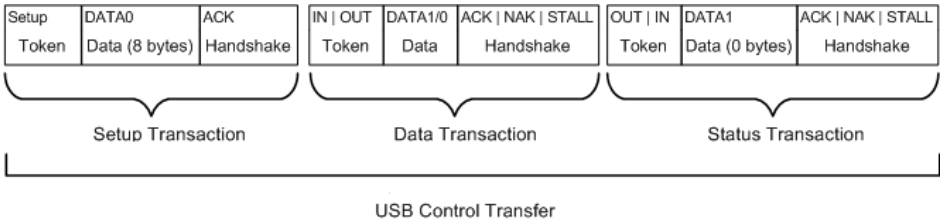


Figure 2.3: Breakdown of the 3 transactions that constitute a control transfer [7]

The setup packet is 8 bytes long and contains information such as : `bmRequestType`, `bRequest`, `wValue`, `wIndex`, and `wLength`. `bmRequestType` and `bRequest` are single bytes and the other 3 fields are 16-bit little-endian values (e.g. if `wLength` = 8 (in decimal), then `wLength` = 0x0800).

It is important to remember that the hardware used in the guitar controllers are the same since RB2. Therefore, we expect that the control URBs found during this project will apply for all guitar controllers. It is however possible that the control URBs differ between platforms and more sniffing must be performed with other guitars on other consoles to validate. Again, proving this hypothesis is beyond the scope of this project.

Chapter 3

Sniffing the USB Traffic Using the Beagle USB 480

In this chapter, the hardware setup is presented and the attempts at sniffing data are performed. Obstacles encountered and solutions to those obstacles are discussed thoroughly in order for the data collection to be repeated if required.

3.1 Hardware Setup

Before any test is performed, it is necessary to verify that the hardware and software function as they are supposed to. Therefore, the Wii USB dongle is plugged directly into the PC and setup to work within the emulator. As confirmed by the Wiki page [8], the pro-guitar is supposed to function using the USB passthrough. The steps from the Wiki are followed for Linux and the pro-guitar tutorials were performed to confirm functionality. Knowing that the hardware and software operate, the next step is to prepare for data capture. Figure 3.1 presents the hardware setup used for data sniffing.

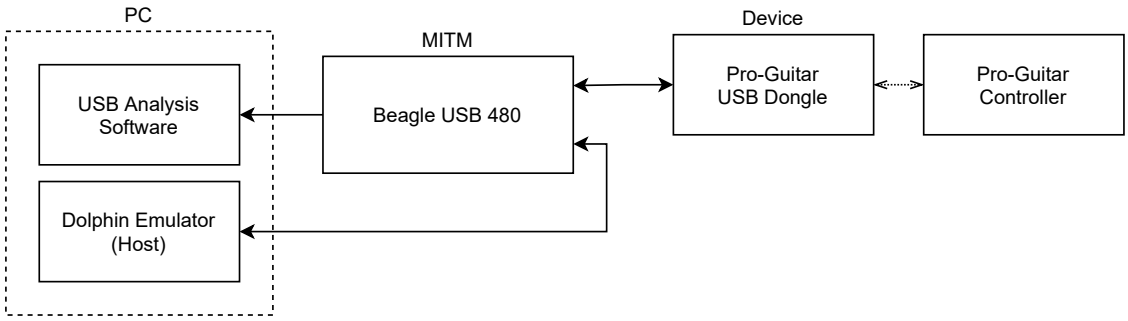


Figure 3.1: Sniffing setup using the Beagle USB 480 as a man-in-the-middle (MITM)

The Beagle 480 operates as a MITM where it allows data to flow between host and device, whilst allowing data analysis. With this setup, the analysis PC and host are the same device. Therefore, both USB ports of the Beagle are connected to the PC. One is used for the collecting and analyzing the data sniffed, and the other is as the host for the dongle.

3.2 Attempts to Sniff Data

The data is captured throughout the entire auto-calibration process that is performed within the RB3 game. The process begins with the sound calibration and once that is completed successfully, the video calibration can be performed. The first obstacle was to successfully complete the audio calibration and is discussed in the next subsection.

3.2.1 Failing Audio calibration

During the first attempt to sniff the data, it was very difficult to complete the audio calibration. In fact, the audio calibration failed most of the time. Figure 3.2 presents the displayed message from the game informing the user that the audio calibration failed and therefore the video calibration could not be started.

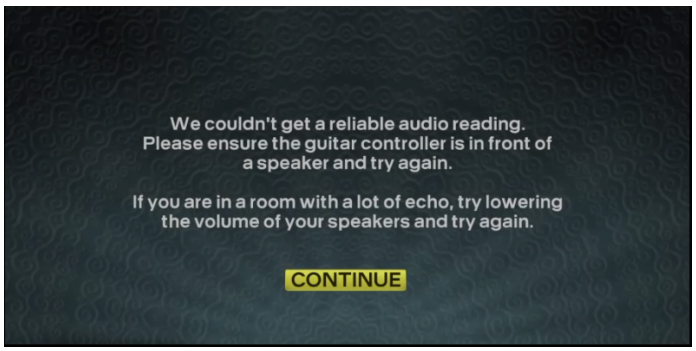


Figure 3.2: Audio calibration failure message

This message suggests that there is echo making it impossible to obtain a reliable audio reading. The first attempt is performed using a 3-speaker setup with two lateral speakers and a central one. It is possible that this multi-speaker setup is causing echo. In order to remedy this situation, audio balance is shifted to one side (left) and the central speaker is unplugged. Another attempt is made using this single-speaker setup.

The second attempt also failed with the same message. Since the message suggested to lower the volume, the volume is lowered and another attempt made. Slight modifications to the volume and the position of the guitar with regards to the speaker were made and the audio calibration repeated. Out of the nearly 80 attempts, the audio calibration only succeeded twice. The first time, data sampling was not started and thus no data was obtained. However, the successful completion of the audio calibration proved that it was in fact possible, but that the process is sensitive. It is possible that the microphone is not functioning optimally due to being nearly 11 years old despite that it is newly unboxed or that the host was not emulating the auto-calibration procedure correctly. Regardless, pinpointing the problem is beyond the scope of this project. The second successful attempt was fully captured and all data from the audio calibration and the video calibration were retrieved. With the data in hand, the next step is to analyze the data and understand what is happening. The following chapter discusses data analysis.

Chapter 4

Analyzing the Data

In this chapter, the captured data is analyzed in order to understand how the auto-calibration process works. The first section discusses the information sent by the dongle (device) to the console (host). The second discusses the audio calibration, while the third section discusses the video calibration.

4.1 HID Report Mapping

In order to analyze the data, it is important to remember that the wireless dongle is recognized as an HID class which means that, upon request, it periodically sends HID reports to the host. These HID reports are 27 bytes in size and contain the state of the device, e.g., whether a button is pressed, a string is picked, the guitar tilted. Most of the mapping has already been done by Jason Harley's work to create a MIDI output using the wireless dongle [9]. Figure 4.1 presents a graphical mapping of the HID reports according to Harley's code and our own testing using `jstest-gtk` and the Beagle 480 analyzer. We confirmed the mapping done by Harley and pushed the analysis further by mapping the D pad vector and testing the range of the Fret and Velocity vectors.

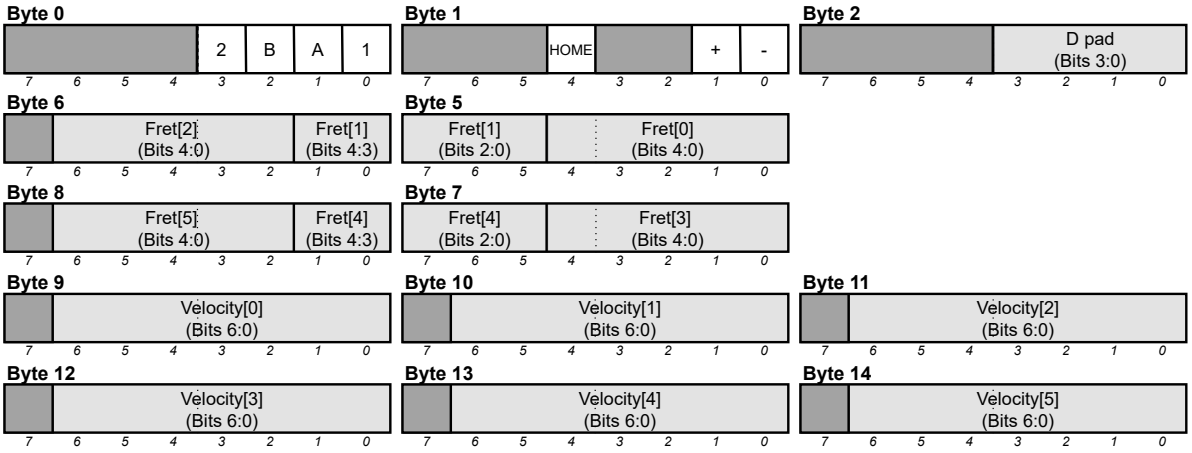


Figure 4.1: Mapping of HID report bytes

A few things are important to note in fig. 4.1. The white squares, in the top row, are single bits that represent buttons. The Pro-Guitar is for the Wii U and therefore uses buttons **A**, **B**, **1**, **2**, **+**, **-**, and **HOME**. These buttons can also be associated to their PlayStation counterparts Cross, Circle, Square, Triangle, Start, Select and PS buttons, respectively. The dark grey represents unknown mapping areas that were not explored and do not hold important information with regards to this project. Also, the light grey areas represent vectors of multiple bits. It is important to notice that the positions of bytes 5 and 6, as well as bytes 8 and 7, are inverted in our representation for clarity. Next, the D pad has 9 possible states represented over 4 bits from 0000 (state 0) to 1000 (state 8). Figure 4.2 presents each

D pad state in the form of bit vectors. Note that at the idle position, the D pad is in state 8.

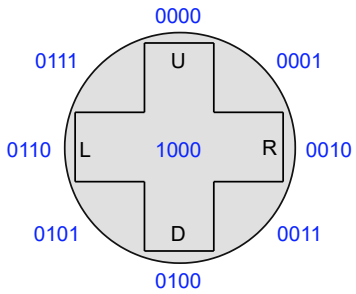


Figure 4.2: D pad mapping

The nomenclature “Fret” and “Velocity” reflect the choices made by Harley in his code and are preserved. The Fret signals are 5-bit vectors ranging from 0 to 17 (in decimal) that correspond to the fret number of a given string. For example, if a player is pressing on fret 17 of string 0, Fret[0] = 0b10001. Similarly, if no fret is pressed, Fret[0] = 0b00000. However, if both fret 3 and fret 17 are pressed on string 0, then Fret[0] = 0b10001. The higher fret number takes priority because it is closer to the pickup which is the device that captures the mechanical vibrations. On a real guitar, the note depends on the length of the string that vibrates and is bound by its shortest distance. The Velocity signals are 7-bit vectors ranging from 22 to 125 (in decimal) that represent string velocity or in simpler terms, how hard a string is plucked. When first powered on, the velocity vectors start at 0. Once a string is picked with enough force to enter the allowed range, it no longer returns to 0 until powered off.

Harley also mapped other features such as Pedal and Overdrive which are not important for this project and are not displayed in fig. 4.1 for clarity. Lastly, we notice through testing that bytes 15, 16, and 17, which were never mapped by Harley, are associated with the tilt sensor. Those three bytes are exactly the same and transition at the same time as the controller tilts. This repetition of tilt data is unexpected at first, but is explained. In fact, byte 15 is used for audio calibration and byte 16 is used for video calibration. The following section analyzes the audio calibration data and the role of byte 15.

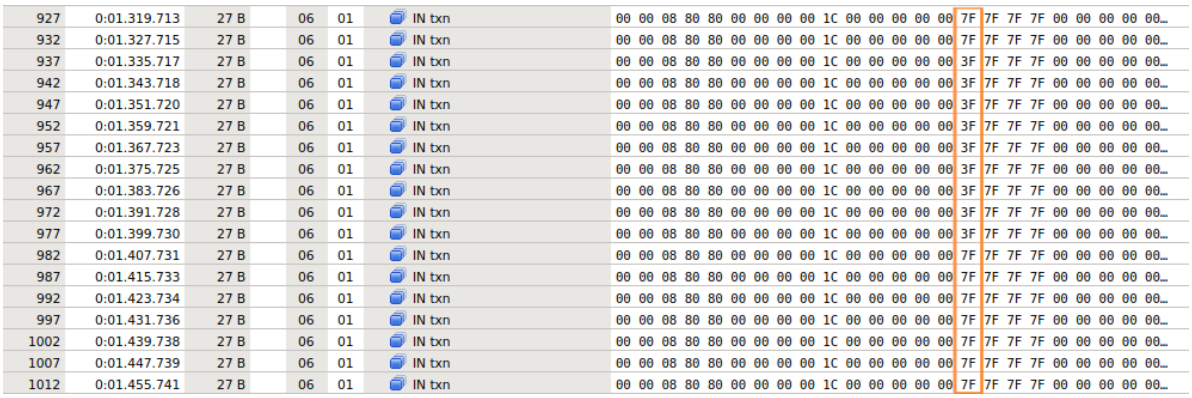
4.2 Audio Calibration Analysis

The next 3 figures are screenshots taken from the Data Center software application which provides information about each URB. The information, from left to right, is the Index, timestamp (m:s.ms.us), length (Len), error (Err), device number (Dev), endpoint (EP) number, type (Record) and HID report data (Summary). The figures are presented in chronological sequence and are the important moments of the audio calibration process. When the process begins, the user is asked to press the **A** button. Based on the mapping illustrated in fig. 4.1, it is possible to quickly seek the byte change that represents button **A** being pressed. In fact, we know that the change is on bit 1 of byte 0 and therefore we expect to see 0x02. Figure 4.3 presents the moment when button **A** is pressed to begin the audio calibration, emphasized by the blue rectangle.

Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
314	0:00.503.547	27 B		06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
319	0:00.511.549	27 B		06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
324	0:00.519.551	27 B		06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
329	0:00.544.627	8 B		06	00	Control Transfer	E9 00 83 1B 00 00 00 02
353	0:00.544.934	8 B		06	00	Control Transfer	00 00 00 00 00 00 00 00
376	0:00.545.140	8 B		06	00	Control Transfer	00 00 00 00 00 80 00 00
395	0:00.545.315	8 B		06	00	Control Transfer	00 00 83 00 00 00 00 00
418	0:00.545.544	8 B		06	00	Control Transfer	E9 01 00 00 00 00 00 00
442	0:00.546.738	8 B		06	00	Control Transfer	00 00 00 00 00 00 00 00
462	0:00.567.560	27 B		06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
467	0:00.575.562	27 B		06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…

Figure 4.3: USB packets exchanged between the host and the device, where HID reports show button **A** pressed (in blue) that triggers the submission of 6 control URBs (in green)

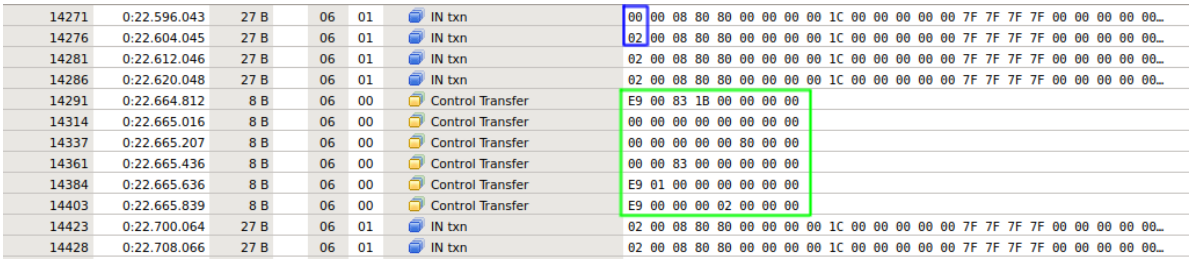
As expected, byte 0 of the Summary column transitions from 0x00 to 0x02 which is when **A** is pressed at index 319 (in blue). We observe that soon after the audio calibration is started, there are 6 control URBs that are sent to the dongle (in green) from the emulated console. As we continue looking at the data, Figure 4.4 shows byte 15 (in orange) of the HID report transitioning to 0x3F. It comes as no surprise that byte 15 is used to contain the information since it was repeating tilt data and was redundant.



927	0:01.319.713	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
932	0:01.327.715	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
937	0:01.335.717	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
942	0:01.343.718	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
947	0:01.351.720	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
952	0:01.359.721	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
957	0:01.367.723	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
962	0:01.375.725	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
967	0:01.383.726	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
972	0:01.391.728	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
977	0:01.399.730	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 3F 7F 7F 7F 00 00 00 00 00…
982	0:01.407.731	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
987	0:01.415.733	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
992	0:01.423.734	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
997	0:01.431.736	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
1002	0:01.439.738	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
1007	0:01.447.739	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
1012	0:01.455.741	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…

Figure 4.4: USB packets exchanged between the host and the device, where HID reports show the chirp being detected (in orange)

The transition from 0x7F to 0x3F and back again repeats periodically. We suppose that these transitions represent the microphone capturing the chirps. The number of transitions are counted and compared to the amount of chirps present during the audio calibration process. The numbers are the same (24 chirps and 24 transitions), and it is therefore logical to assume that our hypothesis is correct. As we continue looking at the data, we notice that the **A** button is pressed again, followed by another 6 control URBs. The **A** button was pressed after the audio latency was calculated and displayed in order to continue to the video calibration. Figure 4.5 shows button **A** being pressed (in blue) and the 6 control URBs (in green).



14271	0:22.596.043	27 B	06	01	IN txn	00 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
14276	0:22.604.045	27 B	06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
14281	0:22.612.046	27 B	06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
14286	0:22.620.048	27 B	06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
14291	0:22.664.812	8 B	06	00	Control Transfer	E9 00 83 1B 00 00 00 00
14314	0:22.665.016	8 B	06	00	Control Transfer	00 00 00 00 00 00 00 00
14337	0:22.665.207	8 B	06	00	Control Transfer	00 00 00 00 00 80 00 00
14361	0:22.665.436	8 B	06	00	Control Transfer	00 00 83 00 00 00 00 00
14384	0:22.665.636	8 B	06	00	Control Transfer	E9 01 00 00 00 00 00 00
14403	0:22.665.839	8 B	06	00	Control Transfer	E9 00 00 00 02 00 00 00
14423	0:22.700.064	27 B	06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…
14428	0:22.708.066	27 B	06	01	IN txn	02 00 08 80 80 00 00 00 00 1C 00 00 00 00 00 7F 7F 7F 7F 00 00 00 00 00…

Figure 4.5: USB packets exchanged between the host and the device, where HID reports show button **A** pressed (in blue) that triggers the submission of 6 control URBs (in green)

The assumption is that the first set of control URBs illustrated in fig. 4.3 activates the microphone which is usually off and allows the sensor to "hear" the chirps and report them back through the HID reports. The second set of control URBs illustrated in fig. 4.5 presumably turns off the microphone. We notice, when comparing both sets of control URBs that only one byte changes. When microphone is activated, byte 7 of control glsurb 1 of 6 is 0x02 and when it is deactivated, the byte is 0x00. These assumptions will be tested in chapter 5. With the audio calibration complete, the video calibration is next.

4.3 Video Calibration Analysis

Similarly to the audio calibration, the 3 following figures are screenshots of the important moments of video calibration presented in chronological order and taken in the Data Center software. The **A** button is pressed to start the video

calibration. Figure 4.6 presents the button being pressed (in blue) as well as the 6 control URBs (in green) that we assume activate the light sensor. We notice that byte 16 becomes 0x00 (in orange) moments after the control URBs are sent. As we saw for audio calibration and its use of byte 15, we see similarly that byte 16 is used during video calibration.

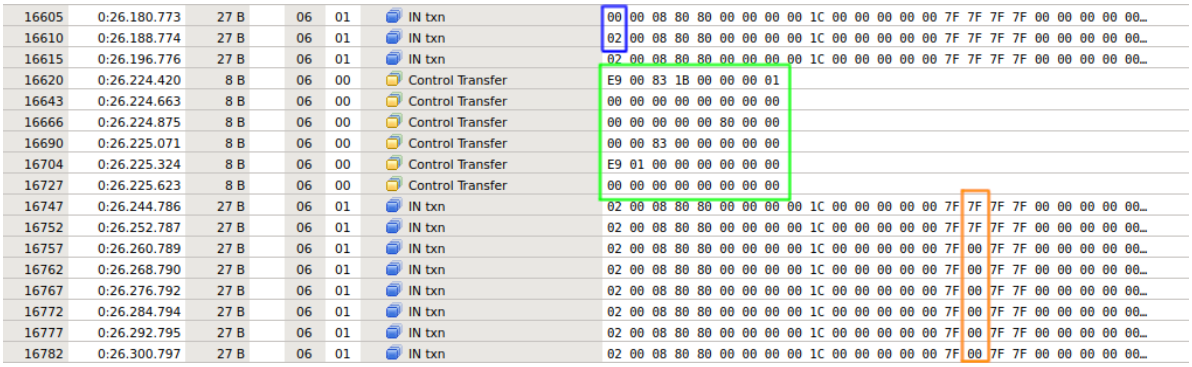


Figure 4.6: USB packets exchanged between the host and the device, where HID reports show button **A** pressed (in blue) that triggers the submission of 6 control URBs (in green) and the initialization of the light sensor (in orange)

Figure 4.7 shows the transition of byte 16 (in orange) that represents the light sensor "seeing" the white screen flash. It is interesting to note that the transition begins at 0x00 which is the initial state (a black screen) and increases gradually through 0x14, 0x28, 0x32, and 0x3C (peak intensity) before it decreases again. This shows that the light sensor measures intensity and may give a false latency if the test is performed in a well-lit room. It is recommended to perform video calibration with the light dimmed.

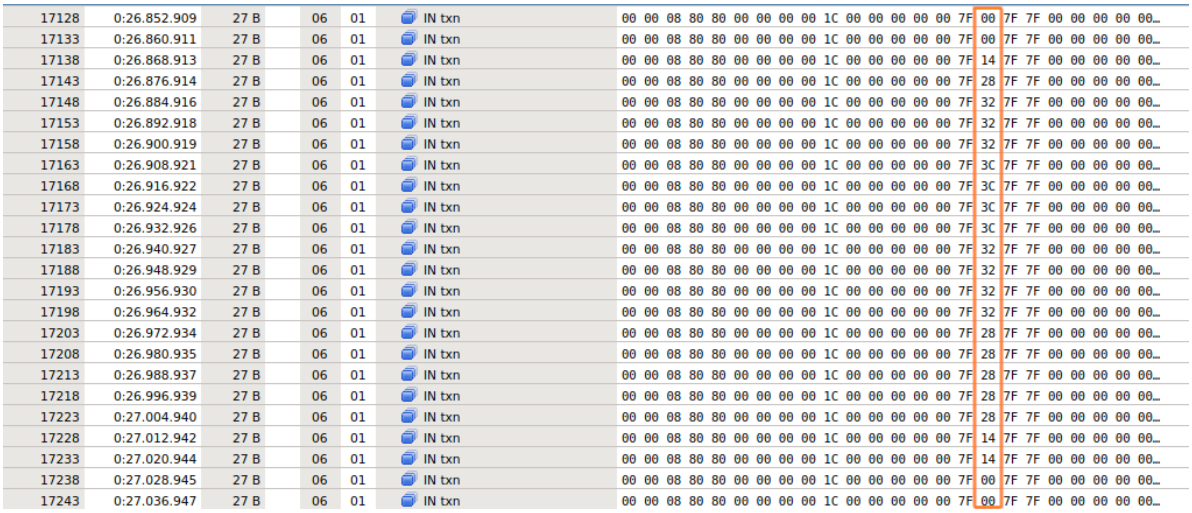


Figure 4.7: USB packets exchanged between the host and the device, where HID reports show the screen flash being detected (in orange)

Contrary to the audio calibration that sampled 24 chirps, the video calibration samples 39 screen flashes in order to calculate the latency. Once the latency is calculated and displayed on screen, the **A** button is pressed to confirm the end of the process and 6 control URBs are expected to deactivate the light sensor. Figure 4.8 presents the **A** button being pressed (in blue) and the 6 control URBs (in green).

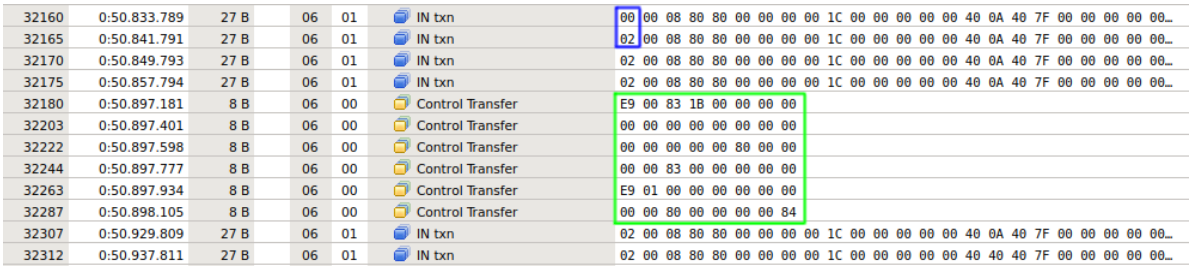


Figure 4.8: USB packets exchanged between the host and the device, where HID reports show button **A** pressed (in blue) that triggers the submission of 6 control URBs (in green)

Similarly to the control URBs seen in audio calibration, when we compare the two sets of video calibration control URBs, we notice that only byte 7 of the first URB changes. It is 0x01 when activating the light sensor and 0x00 when deactivating it. The following section compares the 4 sets of control URBs in preparation of testing the hypotheses.

4.4 Comparing the Control URBs

As mentioned in the previous section, we suspect that the 4 sets of control URBs perform the following tasks : activate microphone, deactivate microphone, activate light sensor, and deactivate light sensor. In this section, we will analyze the setup packets and the data of each URB and compare them.

It is important to note that for each set, the first 5 URBs are OUT types. This means that the data is sent from Dolphin Emulator (host) to the USB dongle (device). The last URB is an IN type which means that the device is sending data back to the host.

The OUT setup packets, as described in section 2.4, are all identical: `21 09 00 03 00 00 08 00`, where

- bmRequestType: 0x21 (Host-to-device, Class, Interface);
- bRequest: 0x09;
- wValue: 0x0300;
- wIndex: 0x0000;
- wLength: 0x0008.

Reminder: The w fields are 16-bit little-endian values.

As previously mentioned, the 5 OUT URBs contain nearly the exact data. When comparing fig. 4.3, fig. 4.5, fig. 4.6 and fig. 4.8, only byte 7 of the first URB changes. That byte dictates which sensor gets activated or deactivated. When the byte is `0x02` (bit 1 is true), the microphone activates. When the byte is `0x01` (bit 0 is true), the light sensor activates and when it is `0x00`, the sensors are deactivated whether it is the microphone or the light sensor. Considering this information, we verified if it was possible to activate both sensors at the same time in order to perform audio and video calibration simultaneously. When sending the activation URBs with byte 7 = `0x03` (bit 0 and 1 are true), neither the microphone nor the light sensor respond to stimuli. Something prevents both sensors from being active simultaneously.

The IN setup packet are also identical: `A1 01 00 03 00 00 1C 00`, where

- bmRequestType: 0xA1 (Device-to-host, Class, Interface);
- bRequest: 0x01;
- wValue: 0x0300;
- wIndex: 0x0000;

- wLength: 0x001C.

The IN URBs data is the response of the device to confirm whether or not it was capable of activating or deactivating the sensors. We notice that when a sensor is activated, the response is all `0x00` and upon deactivation, the response is different. The response data is NOT the calculated latency. We suppose that the latency is calculated by the game. The following section explains what we expect is the complete process and how the latency is calculated.

4.5 Auto-Calibration Process Hypothesis

The auto-calibration process consists of 18 steps as illustrated in Figure 4.9 as well as enumerated below.

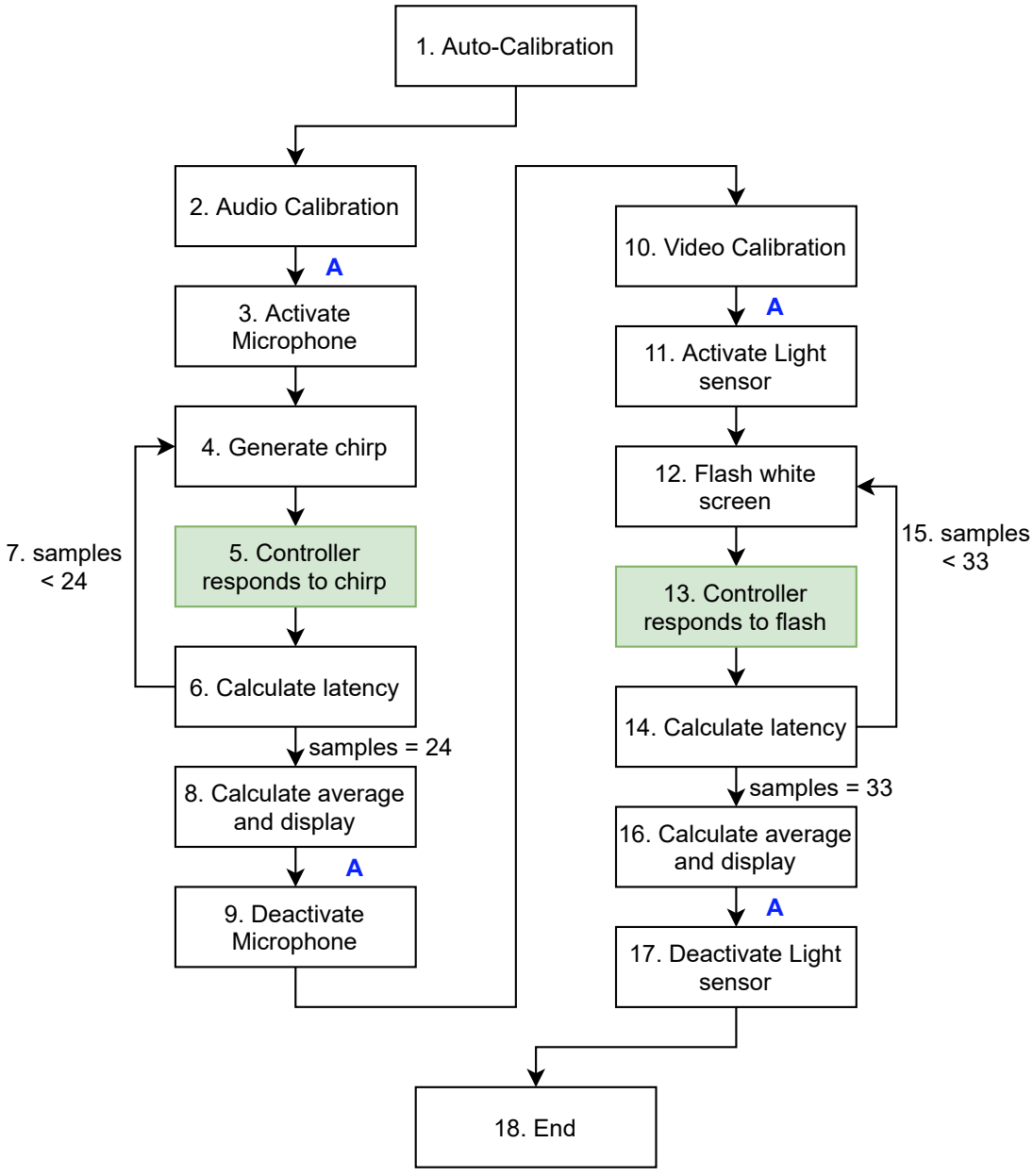


Figure 4.9: Diagram of the hypothesized auto-calibration process

Step description:

1. User chooses to perform auto-calibration and begins with Audio calibration.
2. Game displays audio calibration menu and asks an input button to begin.
3. User presses button to start process which triggers the “Activate microphone” control URBs.
4. Game generates a chirp, saves time of chirp and waits for HID report.
5. Guitar controller “hears” chirp and byte 15 of HID report changes.
6. Game witnesses byte change, saves time, calculates the time difference and saves into an array.
7. Steps 4 to 6 are repeated every 500 ms and 24 times in total.
8. Game calculates the average of 24 samples, displays audio latency and asks user to press button to continue.
9. User presses button to continue to video calibration which triggers “Deactivate microphone” control URBs.
10. Game displays video calibration menu and asks an input button to begin.
11. User presses button to start process which triggers the “Activate light sensor” control URBs.
12. Game flashes a white screen, saves time of flash and waits for HID report.
13. Guitar controller “sees” flash and byte 16 of HID report changes.
14. Game witnesses the byte change, saves time, calculates the time difference and saves into an array.
15. Steps 12 to 14 are repeated every 500 ms and 33 times in total.
16. Game calculates the average of 33 samples, displays video latency and asks user to end process by pressing a button.
17. User presses button to end auto-calibration which triggers “Deactivate light sensor” control URBs.
18. End of process

While analyzing data captures of failed audio calibrations, which were causing issues and preventing continuation to the video calibration, we noticed that the HID reports showed missing chirp detection despite the chirp being generated. It is possible that audio calibration fails when a certain amount of chirps are missed. Our findings show that if 3 or more chirps are missed, the audio calibration fails and displays the screen shown in fig. 3.2. With the captured data analyzed and the hypothetical process present, it is now necessary to test. The next chapter discusses the necessary tests performed to validate the hypothesis.

Chapter 5

Testing the Process

In this chapter, the hypothetical process is validated in two parts. First, the 4 sets of control URBs are individually tested in userspace in order to prove that they do in fact activate and deactivate the sensors. Then, a full program that mimics the auto-calibration process and calculates the latency is created as a proof of concept (PoC).

5.1 Unit Tests for Control URBs

Before creating a full program as proof of concept, it is important to validate parts of the hypothesis. More specifically, the sets of control URBs that are supposed to activate and deactivate the sensors. The activation and deactivation is tested in C language using the libusb library [10] which is a cross-platform user library to access USB devices and the Beagle running as a MITM.

Before activating the sensors, tests are performed to verify that the guitar controller does not respond to chirps nor to flashes of light in its initial state. These two tests are performed as follows. In order to test the microphone, a sample of the chirp is required. Using OBS [11], a short video clip is recorded of the chirp sound during the auto-calibration process in RB3. Then, the sound clip is trimmed using Audacity. This way, the chirp can be played manually. To simulate the screen flashes, a flashlight is used. With the Beagle setup for MITM, the `jstest-gtk` program is run in order to monitor HID reports. The chirp is played and there is no change on byte 15. The flashlight is shined on the light sensor and there is no change on byte 16. Using libusb, a code sends the set of control URBs to the dongle to activate the microphone. The chirp is played and changes can be seen on byte 15. The deactivation set is sent to the dongle and chirps replayed. There are no more changes to byte 15. The light sensor activation set of control URBs are then sent to the dongle. As expected, byte 16 changes when light is shined onto the sensor. The deactivation set is sent once again, light is shined on the sensor, and no changes detected. This confirms our hypothesis about the nature of the control URBs that activate and deactivate the sensors. This information is key to being able to calculate the audio and video latency. With the control sets validated, the next step in testing is to create a proof of concept and this is discussed in the next section.

5.2 Proof of Concept Program to Reproduce In-Game Auto-Calibration

In this section, a proof of concept is created in the form of a program that mimics the complete auto-calibration process. The program must be able to generate the chirp, cause the screen to flash white, monitor the HID reports of the device in order to detect when the cues are sensed, and calculate latency. The following subsections will go over the framework used, the software implementation and the test results.

5.2.1 Software Implementation

The Qt framework [12] was chosen to create the proof of concept due to its ease of development and the availability of documentation, tutorials, and how-to videos. Qt uses the C++ language which allows integration of the unit tests written in C using the libusb library and makes the creation of a GUI simpler.

The software implementation is done in 3 parts : the GUI, the `guitar_control` module and the calibration thread. The GUI is the main thread that displays all instructions to the user. The calibration thread runs in parallel to the GUI thread and handles each step of the calibration process including the activation and deactivation of sensors using the `guitar_control` module, the generation of audio and video cues, and the responses to the cues retrieved through USB. Each part is briefly described below and the complete code can be retrieve at the GitHub repository [13].

The GUI

As illustrated in fig. 4.9, steps 2, 8, 10, and 16 correspond to the GUI since those steps display the instructions to guide the user throughout the auto-calibration process. Step 12 is also performed using the GUI, where the screen flashes by changing the background color from black to white and back to black. Figure 5.1 presents the GUI.

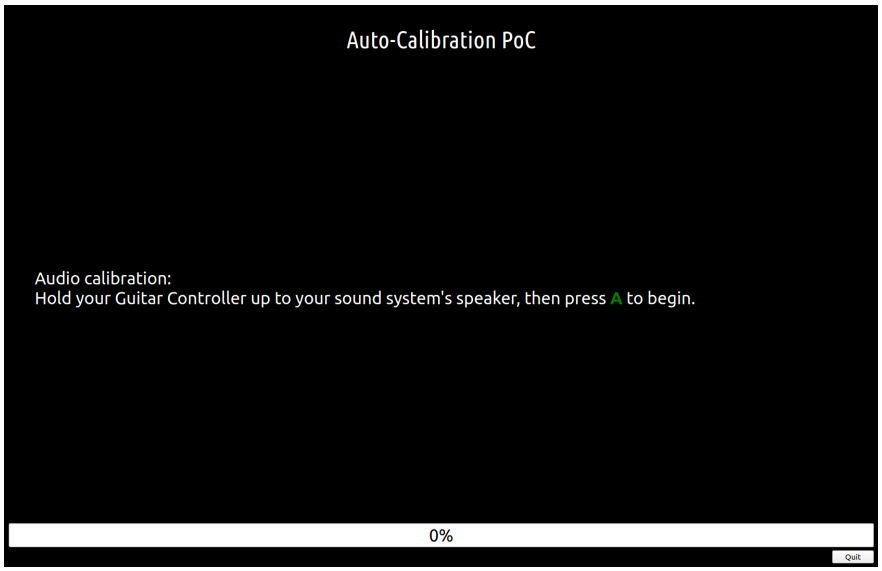


Figure 5.1: the GUI created using Qt framework

The Guitar_Control Module

The `guitar_control` module is completely written in C language and uses the libusb library and the data retrieved in the control URBs sets (fig. 4.3, fig. 4.5, and fig. 4.6 to activate and deactivate the sensors. Three public functions are provided by this module : `mic_on()`, `light_on()`, and `turn_off_sensor()`.

The Calibration Thread

The calibration thread is present at all steps illustrated in fig. 4.9 since it controls the GUI, accesses the USB device and performs the calculations. Figure 5.2 graphically explains how the calibration thread controls the GUI, uses functions from the `guitar_control` module, and accesses the wireless dongle.

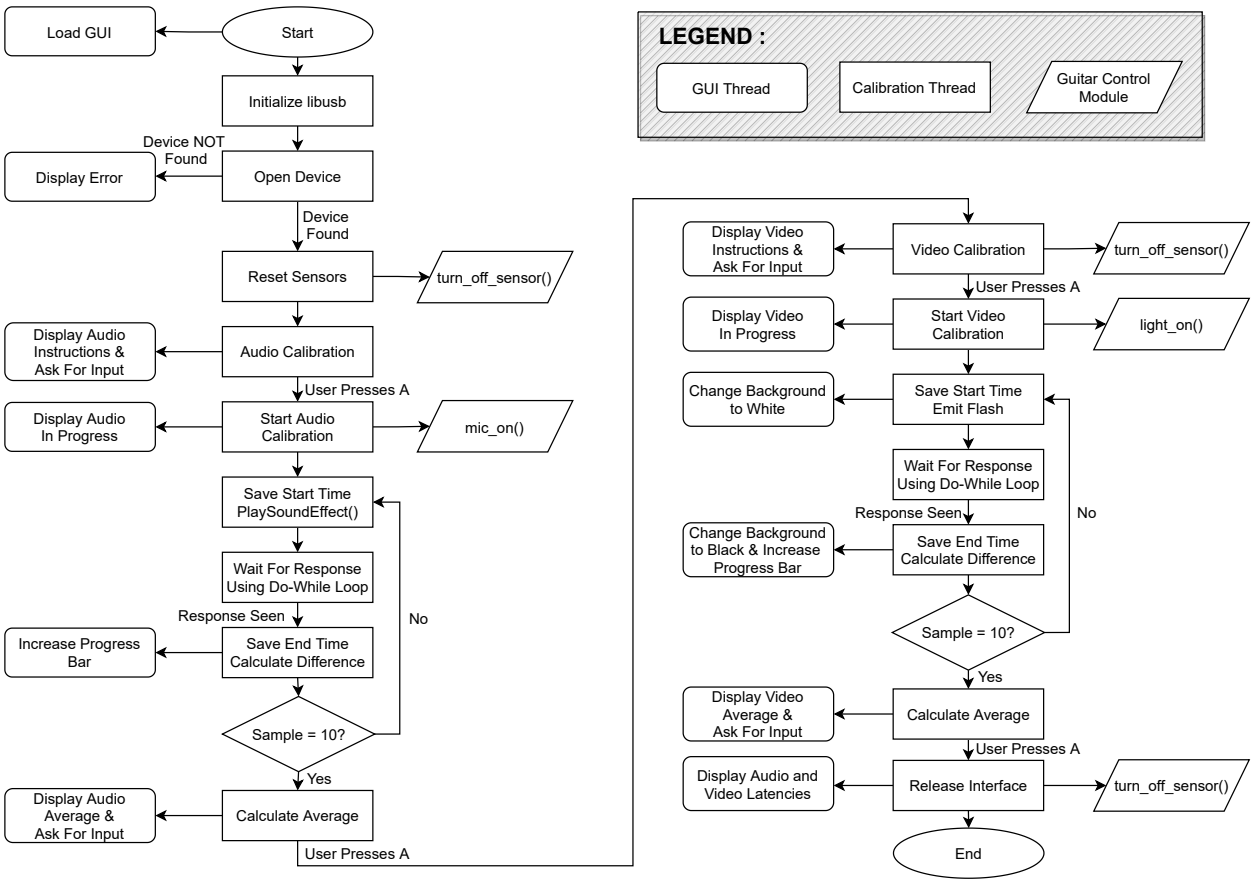


Figure 5.2: Flowchart of the complete PoC process

5.2.2 Test Results

A video demonstration of the PoC is hosted on YouTube [14] and validates our process hypothesis illustrated in fig. 4.9. The program is designed to reproduce the auto-calibration process by generating cues, retrieving the responses to those cues by the guitar controller sensors, and calculating the latency between generation and detection. However, inconsistencies with the latency readings are noticed. In fact, when running an auto-calibration test requiring 30 samples, there are often inconsistent values that are too low compared to the others. These inconsistent values are more present during audio calibration compared to video calibration. During audio calibration, we notice that the majority of the calculated latencies are around 140 ms, but that certain values are inconsistent by being closer to 23 ms. Sometimes there is only 1 out of 30, but other times there may be up to 9 of 30 for audio calibration, whereas video calibration rarely has more than 2 inconsistent values. Table 5.1 presents a test where there are 9 inconsistent values which strongly skew the average during audio calibration and a single inconsistent value during video calibration.

Audio Calibration Sample Latencies (ms)			Video Calibration Sample Latencies (ms)		
158.633007	143.876325	23.859136	110.696888	131.913086	107.808037
191.901229	23.809450	143.975724	107.895114	107.902349	131.873441
167.845131	143.979644	23.745251	107.886857	107.913446	131.869990
23.904819	23.886229	143.941074	107.922037	131.860652	131.886627
143.787850	143.956902	119.813131	107.893265	107.871075	131.874811
23.93116	143.896576	23.859855	131.948929	107.928960	107.902601
119.797258	143.888313	143.937071	107.912994	131.811480	107.922111
119.870079	23.879026	23.909264	107.751512	131.870989	131.977023
167.860203	167.958959	143.939182	83.792729	131.882422	131.814163
119.784581	119.892269	143.885303	107.869224	107.872227	107.882710

Table 5.1: Audio and video sample latencies for a sample size of 30 with inconsistent values (in red)

We suspect that these inconsistent values are due to the implementation of the PoC. The manner through which we poll for HID reports may be causing a report to be queued and waiting to be read which would explain the very short latency. We could explore other strategies to read the /glshid reports with the aim to maximize accuracy. That way, there can be no reports queued. It is also possible that what we consider to be the inconsistent values are in fact the correct latency, while those superior to 100 ms are actually affected by high latency due to using the QtSoundEffect class, although it is the class that offers lower latency compared to the QtMultimedia class. We could explore the use of an audio framework, outside of Qt, that is geared towards low latency. Another possible cause may be due to faulty hardware. Although the guitar controller was brand new in box, the hardware was manufactured over 10 years ago, and the sensors may have become defective over time. In fact, faulty hardware is very likely when we consider how difficult it was to complete audio calibration as mentioned in section 3.2.1. This could be tested by running the PoC with another guitar controller that has no issues with the real in-game audio calibration. Either way, whether the inconsistent values are from the hardware or the software implementation, we expect that when properly implemented into a game, these inconsistencies can be reduced or eliminated.

If the hardware is faulty, then we expect that other hardware will not have the inconsistent values. If the inconsistent values are still present when implemented correctly, the effects may be reduced by changing the algorithm that calculates the estimated latency. The PoC calculates the average of all samples which makes it very sensitive to inconsistent values. A possible solution would be to trim the highs and lows before calculating the average, or to calculate the median instead of the mean.

Chapter 6

Conclusion and Recommendations

Conclusion

In this report, we presented the steps performed to understand the auto-calibration process for the Mad Catz Fender Mustang Pro-Guitar for Nintendo Wii which was unknown until this point. We studied the auto-calibration process through captured data and successfully described and confirmed how the auto-calibration process works, both from a software and hardware point of view. As a result, we have the means to implement the auto-calibration feature into applications such as Clone Hero. At the same time, our understanding of the process enables custom-guitar makers to create hardware that supports auto-calibration and that are compatible with the original games.

Recommendations

1. **Verify that the auto-calibration hardware works the same on other models**
   This project was carried out using the Mad Catz Fender Mustang Pro-Guitar for Nintendo Wii. While the calibration hardware appears the same across controllers, other functional areas have shown us that there can be subtle variations that need to be accounted for, e.g., it's plausible that the sensor activation bits may be mapped to a different location or that different control transfer sets are required to activate the hardware.

2. **Verify if other guitars also show a great variance in the latency measurements**
   We noticed inconsistent latency measurements especially during audio calibration. These inconsistencies appear to be caused by our controller and therefore checking other controllers will help pinpoint the source of the inconsistent values.

3. **Explore other strategies to take the measurements with the aim of maximizing accuracy**
   The inconsistent latency measurements may also be caused by the implementation of the PoC and could potentially be reduced or eliminated by changing the strategies used to measure latency, to generate the cues, and to detect the cues.

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] A. Perry, *What is game mode on your TV and should you use it?* 2021. [Online]. Available: https://mashable.com/article/what-is-game-mode-tvs/ (visited on 02/18/2021).

[3] D. Wilson, *Automatic calibration*, 2008. [Online]. Available: https://www.anandtech.com/show/2648/3 (visited on 10/24/2008).

[4] *Beagle USB 480 protocol analyzer*, 2021. [Online]. Available: https://www.totalphase.com/products/beagle-usb480/.

[5] *Total Phase data center*, 2021. [Online]. Available: https://www.totalphase.com/products/data-center/.

[6] *Dolphin emulator*, 2021. [Online]. Available: https://dolphin-emu.org/.

[7] *How to send a USB control transfer - Windows drivers | Microsoft docs*, 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-control-transfer.

[8] *USB passthrough*, 2021. [Online]. Available: https://wiki.dolphin-emu.org/index.php?title=USB_Passthrough.

[9] J. Harley, *Rock Band 3 PS3/Wii Mustang guitar USB to MIDI software*, 2021. [Online]. Available: https://jasonharley2o.com/wiki/doku.php?id=rb3mustang.

[10] *Libusb*, 2020. [Online]. Available: https://libusb.info/.

[11] *Open broadcaster software | OBS*, 2021. [Online]. Available: https://obsproject.com/.

[12] *Qt | cross-platorm software development for embedded & desktop*, 2021. [Online]. Available: https://www.qt.io/.

[13] D. Nguyen, *AutoCalibrationRB: Proof of concept (PoC) of the auto-calibration process for RB guitar controllers*, 2021. [Online]. Available: https://github.com/dynamix1337/AutoCalibrationRB.

[14] ——, *Auto-calibration proof of concept (PoC) demonstration*, 2021. [Online]. Available: https://youtu.be/3vFgctwwBFQ.