Low Cost USB Trigger Gen With Pi Pico

Charles Wilson

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2024-25

Abstract

The aim of this project is to create a low-cost device with the ability to perform a man-in-the-middle attack on the Universal Serial Bus (USB) and trigger events dependent on certain conditions within the signals. In particular, this project focuses on a Raspberry Pi Pico placed between a keyboard and computer which toggles LED states when specific combinations of keys are pressed. USB is a standard for data transfer and power delivery in virtually every electronic device. However, it is also a medium that is vulnerable to abuse due to the lack of built-in safety measures. As a result, it is possible to get unrestricted access to data being transferred with only a Pi Pico's processing and GPIO capabilities. In this project I show that it is possible to achieve trigger generation for low-speed keyboard signals and I investigate the possibility of expansion to other devices.

Acknowledgements

I would like to thank my supervisor David Oswald for his invaluable guidance, feedback and support throughout this project; I would not have been able to do it without him.

I would also like to thank Sophie Wilson, Rajiv Jayaratnam, Sofia Kover-Wolf, Imaan Iqbal, Fern Warwick and Anna Williams for their advice, feedback and proof reading which enabled me to be confident in my work.

Abbreviations

BCS	British Computer Society
CSV	Comma-Seperated Values
EOP	End Of Packet
FIFO	First In First Out
GPIO	General-Purpose Input/Output
LED	Light Emitting Diode
MITM	Man-In-The-Middle
MVP	Minimum Viable Product
NAK	Negative Acknowledgement
NRZI	Non Return to Zero Invert
PIO	Programmable Input/Output
SDK	Software Development Kit
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

Contents

CHAPTER 1

Introduction

1.1 Project Overview

In this project I created a device that triggers events based on which keys were being pressed on a USB keyboard. This means that when certain keys are pressed in an order, which is equal to a word that has been set as a 'trigger', a specific 'event' will occur. For my implementation the triggers were 'on' and 'off', which caused the onboard LED to toggle on and off respectively. This project also looked at the possibilities of other events that can be triggered using different hardware, as well as what we can set these triggers to be. To be able to achieve this, I had to build a device using the Pi Pico which was inserted between the keyboard and computer, as shown in Figure 1.1, and then write software for this device which would complete the objectives I had set for myself.

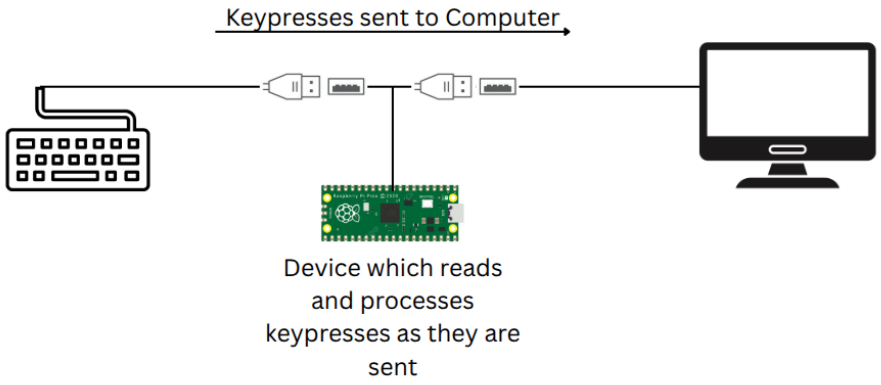

Figure 1.1: Overview of Layout

1.2 Goals of the project

Accurately Read in USB Data Packets: The first goal for this project was to accurately read in and decode USB Data Packets. To do this I needed to use the General Purpose Input/Output (GPIO) capabilities of the Pi Pico hardware. I also needed to write efficient code that could run between each poll by the host device. A typical polling rate for a keyboard is 125 Hz. This means that the processing for each set of packets needed to be done in less than 8 ms to ensure that the device was ready for the next set of packets.

Trigger Generation: Upon obtaining accurate decoded data, the main goal of this project was to generate triggers based upon certain conditions within the data being sent. Within the time frame, I limited this to triggers based upon keyboard data. I thought this would have the most real world applications. This is particularly relevant since there has been a lot of work recently on keyboard 'sniffing' to build upon, such as the Open Hardware Keylogger Implement [21].

No Impact on Normal Activity: I aimed for my analysis of the signals to have no impact on the normal usage of the USB keyboard, as this could cause issues for the user. Possible impacts I wanted to avoid were missing keystrokes or delays to keystrokes appearing on the computer. Disrupting the normal activity of the USB could also produce inaccuracies in the trigger generation, as the packets could be damaged.

Low Cost: I wanted to ensure the project could be reliably reproduced and extended later on. Therefore I set myself the goal that all the hardware I used should be affordable and available from a reputable vendor. If possible, I also wanted it to be more affordable than typical pre-made hardware keyloggers that are currently on the market. I also wanted to do all of the processing on the Pi Pico itself for the same reason to ensure no expensive hardware is required, such as a computer.

1.3 Outcomes of the project

Accurately Read in USB Data Packets: I achieved this goal for when one or two character keys are pressed simultaneously, in addition to modifier keys like 'shift' and 'control'. Three or more character keys being pressed simultaneously was harder to implement due to the amount of comparisons required. Comparisons are done between packets, to see if the keypresses in the current packet are equal to previous packets, but they must be done between each of the six keypress bytes. This situation rarely happens; when users are typing text, typically only one or two characters are pressed at a time. The extra comparisons were required because the keyboard data packets include every key being pressed at that moment, which then need to be compared to previous packets to judge the change in keypresses. I implemented this comparison for when one or two keys were pressed, however the number of comparisons required increases significantly when more keys were being pressed at once; the Pi Pico's processor was not capable of processing this in the required time due to its low clock speed. For typing with many keys pressed at once, the Pico will sometimes interpret

a double press for a key, once when the key is first pressed and another when any of the other keys is lifted. To prevent this, I would either need to find a more efficient way to compare the data in the previous packets, or use a faster microcontroller.

I implemented these readings for low-speed USB. This reflects the majority, but not all, of keyboard use. I did not implement full-speed due to the time taken to pass states through from PIO to C. This forced me to hardcode the length of readings to be taken into the PIO. This hardcoding did not allow for the flexibility of speeds I had initially aimed for, so I opted to implement low-speed due to it being more common for keyboards, as well as being easier to work with.

Trigger Generation: I achieved this goal, however there is room for further development in the future on the impact these triggers can have. This was the most important part of the project to achieve. Currently the Pi Pico can generate triggers based on the keyboard data that we read in without issues (subject to its accuracy). These triggers could toggle the LED on board the Pi Pico, but in the future more advanced hardware such as speakers, or more complex LED matrices, could be added as well to allow for a greater range of effects.

No Impact on Normal Activity: I achieved this goal by connecting the Pi Pico in parallel to the data lines rather than in series. This set-up meant that, no matter the state of the Pico itself, the USB signals still had an unobstructed path to send their data. The Pico doesn't send out any signals over the GPIO pins attached to the data lines, therefore it does not cause interference with the packets.

Low Cost: I achieved this goal as someone could recreate my project using the same hardware I did for £29.10. By excluding the live readout of keypresses that the debug probe provides, the whole project could be recreated at £17.50. At the time of writing, the cheapest keyloggers on Amazon were £39.99 with more compact, wireless, options roughly in the £90-£100 band. My implementation fulfilled all of the features of the low end keyloggers, but is not as compact as the more expensive models.

CHAPTER 2

Background and Related Work

2.1 The Universal Serial Bus

The Universal Serial Bus was first developed in 1994 with the goal of providing a standardised interface between computers and peripherals, as well as increasing the ease of use of expansion ports [23]. The 1.0 specification was released in January of 1996. However, it was only with the release of the USB 1.1 Specification in 1998, which increased the bandwidth to 12 Mbps, that it received widespread adoption. It was included in the iMac G3, and had out-of-the-box support in Windows 98 [12].

For this project we investigated the USB 2.0 specification. This is used for the majority of USB devices such as keyboards and mice [4]. In the 2.0 specification USB devices use a 4-wire cable, as shown in Figure 2.1, that can carry both power and data.

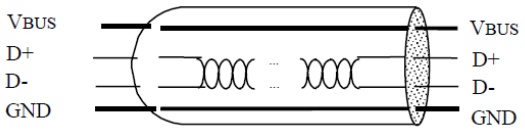

Figure 2.1: Layout of Wires Within a USB Cable [23]

The 2.0 specification covers three possible data transfer rates:

- Low Speed: 1.5 Mb/s
- Full Speed: 12 Mb/s
- High Speed: 480 Mb/s

The signals are encoded using Non Return to Zero Invert (NRZI) with 'bit stuffing' to ensure data integrity and do not require a clock bit. The NRZI encoding means that the data is stored in the change of state of the signals, rather than the current states of the two lines. So, for a USB signal 'a '1' is represented by no change in level and a '0' is represented by a change in level' [23]. Bit stuffing is applied to ensure that the host and device can keep their clock in time. It works by inserting a '0' after every six consecutive '1's before the data is encoded in NRZI. This ensures that there is a transition at least every 7 bits. This stuffing is only applied between the sync pattern and EOP signal. The receiver is then able to decode and discard these stuffed bits before processing the packets appropriately. If it finds any instance of 7 '1's in a row after decoding, it will know there has been a bit stuffing error and ignore the packet [23].

When the data is sent after the encoding, the bits are sent as either 'J' or 'K'. For low-speed signals, 'J' is when the current is sent through the D- line and a 'K' is when the current is sent through the D+ line, this is reversed for full-speed and high-speed signals [23]. This means that the current should only be sent down one data line at a time. There are some exceptions when a current is sent through neither data lines (referred to as a '0'), such as during the End of Packet (EOP) signal, but there will never be a current sent through both [23]. When first plugged in the USB device will indicate what speed it is by pulling a current down the line that correlates to 'J' until the host sends a signal.

Since both data lines are used to send a signal, this means that the USB is considered to run in Half-Duplex, which means it can only 'exchange data in either direction, one direction at a time' [17]. To avoid the host and device attempting to send packets at the same time USB works as a polling bus, where the host will send out a packet asking the device if they have any data to send. If the device does, it will send back a packet containing the information it wants to send, to which the host sends back an acknowledgement. If it does not have any data to send, it will send back a negative acknowledgement. This system ensures mutual certainty that the entire communication has been received correctly by both sides.

Every packet must start with a sync signal made up of 'KJKJKJKK', or '00000001' once NRZI decoded. This synchronises the timings so that the receiver of the signal can accurately read in the rest of the packet. The packets all end with an EOP signal made up of '00J'. The contents of the packet are sent between both signals. Most relevant to my project is the data packet structure. The data packets have three core elements: a Packet Identifier (PID), data bytes, and a Circular Redundancy Check (CRC). Upon retrieval of the packet, the CRC is used to ensure that all the data bytes have been sent correctly by performing a checksum on the previous bytes; if the check fails the packet is rejected.

The maximum number of data bytes in a packet depends on the speed:

- Low-Speed has 8 Bytes
- Full-Speed has 64 Bytes
- High-Speed has 1024 Bytes

For this project I focused on low-speed USB signals, so 'J' will be when the current is sent through D-, 'K' will be when the current is sent through D+, and there can be up to 8 bytes in the data block.

For a keyboard, these 8 bytes are split into the following groups:

- First byte for current modifier keys, such as 'shift' and 'control'
- Second byte as a spacing buffer
- Last six bytes each store a character key that is being pressed.

This means that a low-speed keyboard can send six characters being pressed at once, in addition to the modifiers.

Throughout the creation of the USB specification there was an oversight of the risk of abuse [13], therefore there were minimal security measures put in place. For example, using a device known as a USB Killer [2], it is possible to draw power from the USB port of a computer and then send a powerful electrical pulse back into it, damaging key components such as the motherboard. This is made possible due to the lack of checks on how much power a device should be drawing. This was crucial for my device, since it meant I could run my device off the USB power without the requirement for a separate power source. However, the main security oversight that I exploited in this project was the lack of encryption on the signals being sent through the wires, allowing me to directly read the contents of each packet being sent over the data lines with ease.

2.2 Pico Logic Analysers

The first area of related work that I wanted to look into were implementations for logic analysers that run on only the Pi Pico. I felt that these were the most useful to my project as what I made was in essence a logic analyser with extra capabilities built on top of it, so it was essential I understood the basis before I could build upon it.

2.2.1 Pi Pico Example: Logic Analyser

I found an example of a logic analyser within the 'pico-examples' repository by The Raspberry Pi Foundation. The 'logic_analyser' example [16] offered a program written in C and PIO which acts as a logic analyser. This could read in electrical signals using the pins and turn them into a readable format and then output this over Universal Asynchronous Receiver-Transmitter (UART) to a terminal. This was a promising start of my project, as the reading and output calculations were being done on board. However the output was very

limited, printing '_' and '-' to represent the low and high power states of the two data lines respectively. This made the output of the program very hard to work with, as it was only stored as ASCII within a terminal output. It also offered no decoding, despite requiring a computer to be connected. Despite this drawback, this implementation provided strong evidence that the baseline of my project: reading in the electrical signals using PIO, was possible. I used this work as a base during my own implementation to build upon it by including features, such as activating the reading function upon a change in state and taking readings at fixed time intervals using PIO.

2.2.2 Pi Pico With Pulseview

I then moved on to reading "Using a Raspberry Pi Pico as a Logic Analyzer with PulseView" [11], which was built on the logic analyser example discussed above. In this article, Mark Komus made an improvement so that instead of printing the power states of the lines in text over the serial port, he got the Pico to output the data in a CSV format which he could copy and save as a file to his computer. He could then open the saved CSV in Pulseview, an open-source logic analyser software, which allowed him to decode the I2C protocol that he was focusing on. This brings more capabilities to the implementation, as the readings can now be transferred into a program that is capable of decoding protocols. This would be necessary for generating triggers on the USB data, but it required saving files to a computer and opening them in software, which meant it cannot be done in real time. This limitation was because sigrok, the creators of Pulseview, did not have hardware support for the Pi Pico, and therefore there were no drivers to allow them to communicate directly with each other through the USB it was connected to. However, a Github repository "sigrok-pico" [14] was later created which aimed to introduce real time integration of the Pi Pico's logic analysing abilities into Pulseview. This is done by using the Pico's Software Development Kit (SDK) serial library to communicate through a sigrok driver, cutting out the need for saving and opening files. This repository was then merged into the mainline sigrok repository, officially bringing support to the Pi Pico and allowing full real-time integration into Pulseview. As a result, USB signals were able to be decoded by Pulseview as they were being sent, providing a breakthrough in signal analysis using Pi Pico. The major drawback to this is that this decoding was being done on the computer rather than within the Pico, which requires a specific software and more expensive hardware.

2.2.3 Open Hardware Keylogger Implement (OHKI)

A Github user named 'therealdreg' created a keylogger using the processor of a Pi Pico and an ESP-32, another microcontroller with Wi-Fi and Bluetooth capabilities, which allowed for real time viewing of keystrokes [21]. They implemented this in C, C++, PIO, and Python and the Pico SDK. It had the capabilities to read and decode USB and PS2 protocols using the Pi Pico processor and then used the ESP-32 to host a readout of this data which was accessible over Wi-Fi in real time. Crucially, this demonstrated that the Pi Pico was fast enough to do this level of decoding, which I needed in my own implementation. This work differs from mine by not having the capacity to generate triggers with

the data it decoded from the bus. In addition it used custom hardware, which would not be easily purchasable at a low cost for a normal consumer.

2.2.4 USB Sniffer Lite

Finally, I found an implementation for a USB 'sniffer' that only used a Pi Pico by the user 'ataradov' [3]. In this implementation, written in C and PIO using the Pico SDK, they were able to create a USB sniffer that could read in low and full speed USB signals, but not high speed, as the processor was not fast enough. It is connected via a virtual communication port to a computer, allowing interaction with the Pi Pico. It is possible to tell it when to start and stop sniffing and then print out what it has sniffed. It also offered a variety of settings to choose from, such as a set number of packets to sniff or what speed to sniff. Importantly, the sniffer did not offer any packet decoding capabilities, only identifying the start and end of packets This further demonstrates that the Pi Pico has the capability to do the communication that was done by the ESP-32 in OHKI, even if not over Wi-Fi.

2.3 Keyloggers

Keyloggers are devices that exfiltrate keyboard keypresses through a variety of means [9]. They are generally split into hardware and software keyloggers, where software keyloggers are more common [22]. Software keyloggers work by installing a program onto a computer which is able to read the keypresses from within the operating system, and can be installed through a variety of means such as infected emails [19]. In contrast, hardware keyloggers are a physical device that works by reading the signals from the USB itself, and therefore require physical access to the computer and keyboard [9]. Hardware keyloggers have certain advantages over software keyloggers such as being able to capture keypresses before the operating system starts (such as BIOS passwords) and working on all operating systems. My project aimed to read the USB signals directly, therefore I focused my keylogger research on hardware implementations rather than software.

2.3.1 Inline Keyloggers

The most relevant type of hardware keyloggers to this project were Inline Keyloggers, which complete this task by going between the host and the device on the USB lines. It is most often implemented as a device resembling a flash drive with a female USB connector on one end and a male on the other, shown in Figure 2.2. These are plugged into the back of desktop computers between the keyboard cable and the USB port, where a user would not be able to see it [18]. From here there are a variety of different way these keyloggers handle the data they get. They most often keep the keylogs in large flash storage, capable of storing up to 5 years' worth of keystrokes [7]. They can be triggered to release their contents by a certain key combination being pressed [9], which will output all previous keypresses as text into a document. More complex examples will have network capabilities and can stream the keypresses out over Wi-Fi or Bluetooth. The ability to react to a key combination to release the old data

shows that these have the ability to do trigger generation. However due to the locked down nature of the hardware, it can only be used for release, rather than other more complex purposes.

Figure 2.2: Example of an Inline Hardware Keylogger [6]

2.3.2 Other Types of Hardware Keyloggers

There are also a variety of other types of hardware keyloggers, such as custom firmware that is flashed to the chips on the host itself or within the keyboard [20], which work the same way as inline keyloggers, but take their readings from one side of the communication. Another comparable method is wireless keyboard sniffers, which are able to collect the packets being sent through the air between a wireless keyboard and its receiver. These are often encrypted, which adds a layer of complexity as the signals need to be decrypted before someone can discover the keystrokes. Some more unique methods also exist, such as monitoring the sounds made by the keyboard, allowing the device to tell what has been typed by the subtle differences in the keypress sounds [9]. It is even possible to capture the electromagnetic emissions from a USB cable from up to 20 metres away [9], although this requires complex decoding. These methods each provide certain benefits over inline keyloggers, such as being harder to identify or not requiring direct physical contact with the devices, but have their own drawbacks such as needing to decrypt the transmission or having to edit the firmware of the computer. I chose to make mine most similar to an inline keylogger as it offered the most robust access to the data lines without having to modify existing hardware.

2.4 How Does My Work Differ?

A variety of existing systems are described above, however none of them achieved all of the capabilities that I planned to create. The key distinction is that none of the existing work implemented custom trigger generation, which I successfully achieved, demonstrating that my approach offers a unique contribution to the field. I even achieved similar capabilities to many of the implementations which used expensive custom hardware on my own lower cost hardware.

CHAPTER 3

Legal, Ethical and Professional Issues

3.1 Legal

Before my project I familiarised myself with the Computer Misuse Act 1990 [8], which regulates what you are allowed to do with computers, especially ones you do not have permission to use. I mitigated this risk by only ever using the device on my own computer which I had the right to use. There is a possibility that this project could be used by someone else to steal keypress data, which would violate the act. Section 1.1 states '*A person is guilty of an offence if ... he causes a computer to perform any function with intent to secure access to any program or data held in any computer*' and '*the access he intends to secure ... is unauthorised; and ... he knows at the time when he causes the computer to perform the function that that is the case.*'[8]. In order to mitigate this risk, I made the design decision not to store any data in non-volatile storage. This means once the device is unplugged all keypress data is gone. It also gets overwritten while the program is running, so someone could only get a small number of previous keypresses if they looked at the Random Access Memory while it was running. The only exception to this was that I included a method of outputting the keypresses over UART to a serial terminal running on a computer, which could still be used to watch the keypress data to get access to a computer. However UART is a wired protocol and therefore the attacker would need to run a wire from the device to the attackers computer to be able to watch the keypresses. UART has a recommended maximum length of 15 metres [15], which means an attacker would need to find a way of installing their computer within that distance of the computer they are trying to get access to, which makes the job a lot harder. With careful implementation they could add another microcontroller, like an ESP-32 which was used in the OHKI, however if they have the knowledge as to how to do that they would already have the knowledge to create my work using the ESP-32 alone so I do not think it is a likely scenario. As a result of these design decisions I believe that the risk of a malicious actor using the device to steal keypresses has been adequately considered and mitigated.

3.2 Ethical

Throughout my project I have made sure that I conducted all of my work ethically, by following the Association for Computing Machinery Code of Ethics [1] which guided my ethical goals for this project; most notably it specifies to '[a]void harm', which was very important when creating something that has the capacity to cause harm if used incorrectly. When used for its intended purpose, my project does not violate the guidelines laid out. Instead, it teaches people about the risks associated with USBs and hopefully prevent them from being taken advantage of in the future. This falls under 'Contribute to Society and Human Wellbeing' within the code of ethics. However, I have also considered the possible ethical issues of people using it for unintended purposes. This could violate many of the ethical guidelines, which focus on a person's ability to consent to their data being collected, laid out in 'respect privacy' within the code. To prevent these ethical issues, as laid out in the legal issues section, I took some precautions that made it harder for my project to be adapted to be used for malicious or unethical purposes.

3.3 Professional

During the project I made sure I conducted all of my research and development within the professional guidelines set out by the British Computer Society (BCS) code of conduct[5]. As a student member of BCS it was essential that I followed their guidance on professional integrity which states you must '*have due regard for ... privacy, security and wellbeing of others*'; this section was most relevant because if I did not act with due regard throughout the project I could produce a product which is capable of violating people's privacy. The measures I took were only using it to access my own data, and when getting people to test it out explicitly telling them the purpose of the test and ensuring I clarified that no keypress data was being collected, as is required in the code of conduct. Once I make my work public to people who are not BCS members there is a chance they might not follow this professional guidance, however, the majority will be other computer science students and professionals who will also be bound by the code of conduct.

CHAPTER 4

Project Requirements

4.1 Functional Requirements

Functionally, it is required that a user can place my device between their keyboard and computer, and trigger events upon certain keypresses. In order to achieve this requirement, there are certain sub-requirements:

- **Interfacing:** The device can interface with the USB port so that it can retrieve data signals.
- **Inputting:** The device can read and decode keyboard data packets containing modifier keys and character codes for both low-speed and full-speed signals.
- **Triggering:** The device can trigger events on certain conditions within the signals, such as toggling an LED state.
- **Powering:** The device can run independently using the power from the USB port, without external power.

The above requirements combine to form the core structure of this project.

4.2 Non-Functional Requirements

I also set out some non-functional requirements for myself:

- **Accurate Identification:** 95% accuracy on new keyboard character codes being identified from the bus, so that I have the correct data being used for triggers and avoid miss-fires.
- **Accurate Triggering:** 100% accuracy on event triggers, assuming the underlying data is correct.
- **No Impact:** 100% accuracy on normal typing with the keyboard, and no errors raised by the host computer.
- **Low Cost:** All of the components of the device must be affordable and easily purchasable so that others can recreate it. This also makes it easier for me to replace broken parts.

4.3 Hardware and Software Constraints

This project was subject to both hardware and software limitations during its development, due to my choice of components. These influenced my design decisions and implementation.

Hardware Available:

- Raspberry Pi Pico
- Breadboard for Pico
- Male Header Set for Raspberry Pi Pico
- 120-piece Ultimate Jumper Bumper Pack (Dupont Wire)
- USB 2.0 Male Type A Plug Connector Breakout Board - CIE-YY27
- USB 2.0 Female Connector Breakout board - CIE-YY24
- Raspberry Pi Debug Probe

Hardware Constraints:

- **Processing Power:** The Pi Pico has a clock speed of 133 MHz [10], significantly lower than modern computers, which limits the amount of calculations it can do in a certain amount of time. This constraint meant we needed to write efficient code to ensure the Pi Pico can handle it.
- **Limited Memory:** The Pi Pico only has 264KB of onboard memory, which constrains how many previous keypresses we can store. It also means we need to guarantee there are no memory leaks which would quickly cause the program to fail.
- **Limited Storage:** The Pi Pico only has 32MB of of onboard storage, which limits how many files we could write to storage if needed in the software.

- **USB Polling Rate:** The USB polling rate can range from 125 Hz to over 1000 Hz [24]. This means we have a time constraint to run our program in the short time between each poll.

Software Constraints:

- **Language Support:** The Pi Pico SDK only supports C or micropython when writing programs for it, which means I need to write my software in one of these two languages.
- **PIO Limitations:** Each PIO state machine can only support up to 32 instructions, which limits the complexity of possible programs writtin in it.

CHAPTER 5

Design

5.1 Hardware Design

The core of my hardware was a Raspberry Pi Pico; I was confident it would fit my needs as I had reviewed work that demonstrated its capabilities. I then needed to use USB male and female breakout boards to separate out the data and power lines for my usage. From there I wanted to place the device in parallel with D+ and D- connected to two of the GPIO pins and VBUS and GND connected to VSYS and GND respectively. This allowed the Pi Pico to use the power directly from the USB port as well as easily access the data lines without needing to cut into the cable. This made it safer and more reliable than other methods which could have risked damaging the computer or keyboard.

5.2 Software Design

Once the Pico was connected to the USB wires, I then needed to run the processing on board. I expected this to follow the basic structure of:

1. Identify change in state to signal start of a packet.
2. Read in full communication stream using PIO.
3. Identify the data packet within the stream.
4. Separate out each byte of data.
5. Compare to bytes from previous packets to prevent repeats.
6. Add key presses to array of previous keypresses.
7. Compare keypresses to the set trigger.
8. If they are equal then trigger the set event.

I planned to write this program in C and PIO, as the Pico can only run C and micropython natively, and micropython is too slow for what I wanted to achieve. All of the related works are written in C, indicating that this is the preferred language in the field currently and a strong language to choose. I had to use PIO as it allowed direct programming of the GPIO pins for the Pi Pico.

Alongside the core process cycle I have described, I also wanted to have a method of outputting the recorded keypresses to a terminal, acting as a more traditional keylogger would. This made testing, debugging, and adding extra functionality to the device easier as it showed what had set off the triggers.

CHAPTER 6

Implementation

6.1 Project Management

Before I started working on the design and creation of my project, I made sure that I understood the current state of the field, so that I knew the limitations of what already existed, as well as ensuring I was not creating something that had already been produced. Early on through this research I realised that PIO would be an essential aspect to enabling the speed required for the project, so I learnt the basics of the language alongside the research into related works. Once I was satisfied with my understanding of the field, I created a draft of what I would like my implementation to achieve (see section 3.1). From there I laid out each step I would need to implement and added these to a Kanban board to keep track of my progress. Due to the nature of the project, I chose to create a minimum viable product (MVP) of my hardware first, as this would allow me to test my software as I developed it, rather than having to wait for the finished hardware. The project timeline and milestones can be seen fully in Figure 6.1.

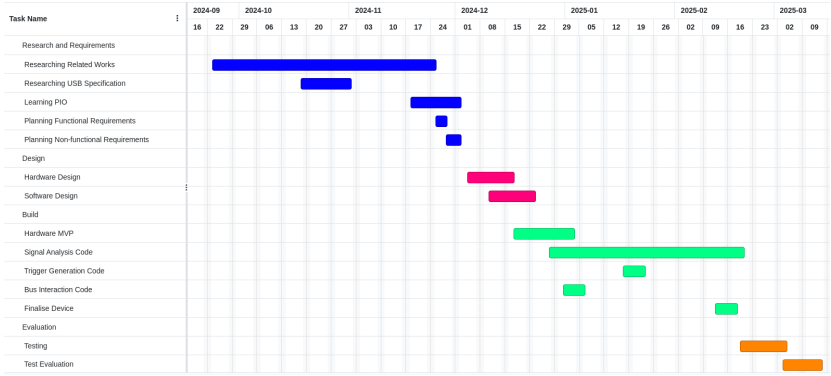

Figure 6.1: Gantt Diagram of Project

I attended fortnightly meetings with my supervisor throughout the project to ensure that I was on track with my work, as well as ask any questions I had. I also used Git to version control my software, ensuring that I could roll back any changes that broke my software, which was essential given the software could only be tested once imaged onto the Pico and the old code had been wiped.

6.2 Hardware Implementation

I followed the hardware design I set out and implemented it using a Rasberry Pi Pico with headers, a breadboard, 8 Dupont Wires, 2 USB breakout boards and a Raspberry Pi Debug Probe. All together, the cost of this hardware came to £29.10. I then combined these components as planned in the design stage, by wiring the separate wires in the USB into their respective GPIO pins on the pico, shown in Figure 6.2.

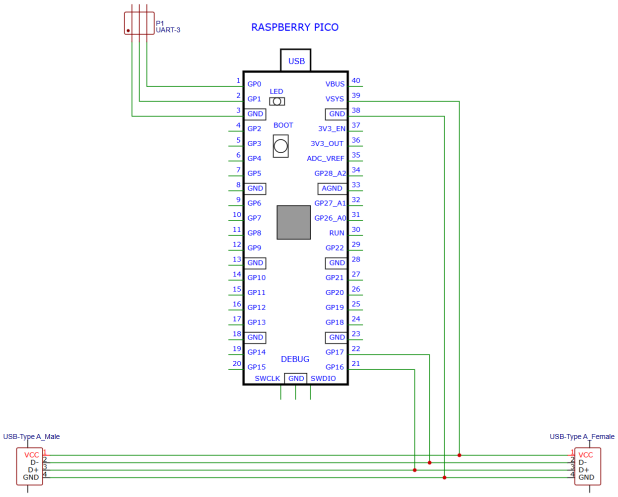

Figure 6.2: Wire Map of Hardware, showing which GPIO pins were used

In my original design, I had planned to have the terminal output directly through the Pico's USB port, however this posed issues with the power delivery, as it was drawing both from the power lines connected in parallel as well as the USB port it was plugged into. As I wanted it to work with or without the readout, I could not remove the power lines, so I had to find another way to connect the terminal output that did not provide power. I managed this with the Raspberry Pi Debug Probe, which can do UART to serial terminal conversion, providing the output I required without causing issues, as the Pico had native UART output.

One aspect of the hardware implementation that could have been improved was the output devices for the triggers, as I only have the singular LED built into the Pi Pico itself. If I had more time, I would have liked to add another form of output, such as an LED matrix, as this would allow me to create more unique outcomes that can be triggered.

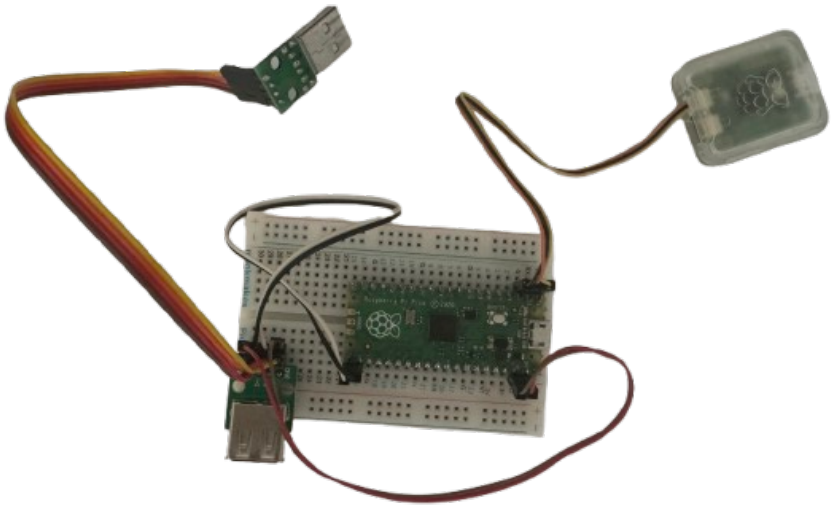

Figure 6.3: Photograph of Finished Hardware

6.3 Software Implementation

I implemented the software in C as planned, but had to rely on PIO a lot more than originally thought. I had planned to use the PIO just to pass through the GPIO pins to C at each interval to be analysed, which would have allowed better identification of start and end of packets. During the development, however, I found that this planned implementation did not work as it took too long to transfer the state between the two. This meant I could not capture enough readings to get an accurate measure of the changes in states, which further meant I often missed the start of the signals or got inaccurate data packets. Therefore I had to use PIO to internally identify the starting change in state, and then read in a fixed number of readings which it transferred as a block to the C code. The downside to this was that it required hardcoding a fixed number of readings to take. This meant it could only be used for a preset length of message. I chose to have this be the length of a low speed USB keyboard communication, as this was the most important and commonly used standard. This allowed the flexibility to set how many readings I wanted per bit being sent, which I set to roughly twenty, to make it easier to identify exactly how long a certain state was held for. The device parsed this imported block to identify the readings that had data packets by finding the packet identifier. If it is a 'NAK' packet then the device can ignore it, but if it is a data packet it can split the next eight bytes into separate variables which store their values.

The next step was comparing the bytes to previous packets to prevent repeats which posed another complication; By pressing a key, it will be logged for multiple polls (which are every 8 ms), therefore in order to prevent the key from being logged multiple times, the data must be compared. I had initially assumed we could prevent this by comparing the whole packet with the previous packets to see if they were the same. This worked well when one key is being pressed at a time, as I could just see if the bytes in the last packet are equal to the current one. During the development I realised that this did not work for when multiple keys are pressed, which happens often during typing, as when one key is lifted, the entire packet appears different to the last, even if all other keys within the packet are not new, leading to them being added multiple times. To resolve this, I tried to implement piecewise comparison between each byte of the new packet with each byte of the previous packets. However the number of comparisons required for this implementation was too large for the Pi Pico. This meant my program would take longer to compare the bytes of a packet than the gap between the packets, causing me to miss keypresses. Due to the time complexity I rewrote my code to compare the first two keypresses present in a packet. This made it perfectly accurate for if one or two keys are pressed, which covers a typical 100 words per minute typing speed. However when more than two keys were pressed in explicit tests there were often repeat characters. This was a limitation I had to make to ensure that normal usage worked smoothly, even if it meant edge cases were not as accurate.

Once I had identified which key presses were new, I added these to an array that stored n previous keypresses in memory, where n can be set by the user. The array works as First In First Out (FIFO) so that stale keypresses are not stored unnecessarily which ensures the Pico's small amount of memory does not become full. The device also stored a pointer to the most recent value in the array so that it can easily look at it. Once a new keypress is added, it used the value of the pointer to compare the trigger strings to the new keys, working backwards by comparing the last letter of the strings to the most recent value, then if they are equal it will compare the penultimate letters to the keypress before. It continues until a full trigger is found to be equal, which causes the associated event to occur, or if none of the triggers are equal, it moves back to waiting for the next packet. Currently the triggers need to be written into the program before it is flashed to the Pico. The Pico is set to 'on' to turn the LED on and 'off' to turn the LED off, but it is noted that more settings could be added.

The full codebase is released under the GPL 3.0 open source license and can be found at:
`https://git.cs.bham.ac.uk/projects-2024-25/cxw304`

6.4 Project Applications

By developing this project, I aimed to highlight the vulnerabilities within the USB protocol that can enable data exfiltration. The intended use case is for others to replicate my work in order to observe first-hand how USB signals can be exploited, particularly through testing custom trigger generation. For instance,

users could set their password as a trigger and monitor how often it activates, thereby demonstrating how frequently a malicious actor might have captured it.

The project could be developed to include an LED matrix, which builds upon the current singular LED by allowing for more colours and being able to show a wider variety of things. One potential use for a LED matrix is to have it flash when a certain combination of keypresses are detected (such as commonly misspelled words) acting as a form of real-time feedback during typing. Where this is unique is that it works universally on any operating system and program. Compared to app-specific spellcheckers, this will work no matter where you are typing into.

Alternatively, for a more aesthetic application, different colours could be triggered by different letters, creating a visually engaging typing experience where each keystroke lights up the matrix in its own unique hue. Visually striking setups have become a trend among gamers and coders.

I also recognise the potential for misuse by individuals seeking to exfiltrate data. To mitigate this, I intentionally designed the system around real-time analysis and trigger detection without storing any captured data. This approach reduces the risk of the tool being misused for malicious purposes.

CHAPTER 7

Evaluation

7.1 Testing Methodology

To test whether I had achieved the requirements that I had set out for myself I created some testing methodologies. For the requirements that had a percentage accuracy, I devised a method where I would type out 1000 pre-set keypresses, including every relevant letter, number and symbol, and then compared what I had pressed to the output presented to the terminal through the debug probe. I then repeated this twice more to ensure the accuracy of the results. I removed the factor of mistypes by having them being written into a text document during the test. Therefore when comparing the device output where a mistake was identified I could confirm if this was also present in the text document, and thus due to human error. I only conducted these tests on low-speed USB, as I had not implemented full-speed. To test whether I had achieved my low-cost aims I researched other similar products online from a variety of sources to get a baseline for how much those cost. Other requirements were tested passively during general usage, such as the ability to run independently from power and interfacing with the devices.

7.2 Functional Requirements

I achieved all of the functional requirements for this project, however certain requirements were only possible in limited situations, so could not be implemented for all possible conditions.

- **Interfacing:** I succeeded fully in having a seamless interface between the USB keyboard, the device and the host computer. The USB breakout boards allowed my device to be connected to the other sections using standard USB connectors, making it intuitive to use.
- **Inputting:** I was only able to program my software to read in and decode low-speed USB signals. I did not implement full-speed as the Pi Pico was

not fast enough to consider what speed the USB is running at and therefore I had to make a choice which speed I wanted to implement. I chose to only do low-speed as this made up the majority of devices, as well as being easier to implement as they are slower and provide more time to analyse, which was necessary with the time constraints imposed. This naturally limited the devices it can be used for. All of the keyboards I used during testing only used low speed so this was not an issue for my purposes, but others who wish to replicate my work might not have the same experience. This limitation allowed me to easily identify and copy the data byte values due to the lower speed. For low-speed it can identify and separate the modifier keys and character codes.

- **Triggering:** I implemented all standard keyboard codes, including all letters, numbers, and standard symbols. I chose not to include shortcut keys, such as 'print screen' or multimedia keys, so I could not trigger upon those types of keys. I made this choice as I was storing the previous keypresses as chars in C which only allows one byte per character, meaning special characters would need to be stored as strings and requires doing more string comparison rather than simpler char comparison. I was able to create any triggers made up of the characters laid out in the code and have these be triggered. I would like to find a way to set triggers while the code is running on the Pico, since, in this project, the triggers were written into the code. This would also be promising for the prospect of using this work in any sort of aesthetic commercial context, whereby buyers can set their own triggers.

- **Powering:** I managed to achieve my goal of having my device run solely off the power being sent from the USB port, making it far more practical to use.

In total I managed to achieve almost all of my functional requirements, with the multi-speed capabilities being the only one I was not able to implement.

7.3 Non-Functional Requirements

I also achieved my non-functional requirements for the majority of cases tested.

- **Accurate Identification:** I achieved my first non-functional requirement, having 95% accuracy of identifying the new character codes for one or two keypresses. For this I set my 1000 keypresses to a repeating story which included letters, numbers and symbols to ensure full coverage in the test. For normal usage, when at most two keys are being pressed at one time, I managed to achieve no misreads throughout my 1000 keypress testing, making it 100% accurate in those tests. This was however contrasted by the testing I did in which I ran my hand across the keyboard, which triggered more than two keys at one time, in which the accuracy dropped down to only 64% in a 1000 keypress test. This shows that I did not achieve this requirement in certain situations. In both tests usage of modifier keys worked fully, independently of the accuracy of the character key readings.

- **Accurate Triggering:** I achieved the aimed 100% accuracy of the trigger, given the keypresses being read in were correct, I managed this by setting the 1000 keypresses to be alternating 'on' and 'off', the triggers I had set to turn the LED on and off respectively, and seeing if the LED would change at each iteration. For this I did two sets of tests, one typing at a normal speed, and one where I ensured I hit more than two keys at a time, to see if the results changed between the two cases. The results were consistent between the two where, given the byte reading was accurate, the event was always triggered. I also tested for false positives, whereby I typed 1000 characters of words that were similar to the trigger phrases 'on' and 'off', such as 'of' and 'no', to see if it would trigger erroneously. Like the previous test, this worked flawlessly with the normal speed testing, and correctly for the multi-press tests, once the input errors were accounted for.
- **No Impact:** I achieved the 100% accuracy for normal keyboard activities, to show that my device has no impact on the USB signal itself. In another 1000 keypress test I validated that every key I pressed was received accurately by the computer, demonstrating a strong likelihood that there was no impact. In this test I did not compare to the terminal output and manually checked that I had pressed the right key each time. Then, to ensure this was true, I did it 5 times, and there were no mistakes in any of the 5.
- **Low Cost:** My final requirement to be low-cost was successful; I purchased all the core components for only £17.50, with an extra £11.60 for a text readout of the keypresses. Even with this component included, this still renders my implementation as cheaper than current hardware keyloggers on the market, the most similar purchasable item. These come in at around £40 - £100 on Amazon at the time of writing. I knew that the price had not changed significantly for the last 16 years, since it cost \$50-\$150 in 2009 [18] which is equivalent to £38 - £134, almost the same as the current price.

Non-functional Requirement	Goal	Result	Fully Achieved
Accurate Identification	95%	1-2 Keys: 100% 3-6 keys: 64%	✗
Accurate Triggering	100%	100%	✓
No Impact	100%	100%	✓
Low Cost	$\leq$ £40	£29.10	✓

Table 7.1: Results From Testing on Low-Speed USB

CHAPTER 8

Conclusion

8.1 Outcomes

Within this project I researched current work in the field of keyloggers and Pi Pico logic analysers, and then used this knowledge to design a low-cost device using a Pi Pico that could trigger events conditionally upon the data being sent through the Universal Serial Bus from a keyboard.

My approach offers distinct advantages over the current work in the field, which do not have settable trigger generation, affordable hardware or customisability. While on their own some existing solutions had parts of the requirements, none of them were able to achieve them all, especially custom trigger generation, which did not exist in any current device. As a result my project is novel in the field and offers a variety of practical applications ranging from universal spellcheck to computer lighting control.

During testing I identified that my work had achieved its planned results within the 'normal' usage range, with a low speed keyboard and normal typing. However, at other USB speeds and when lots of keys are pressed at once it could not accurately trigger events, which is an area of improvement for the future.

The key findings were:

- It is possible to read data packets directly from a USB cable.
- The Pi Pico is capable of reading in and decoding low-speed USB signals, but not higher speeds.
- The Pi Pico cannot accurately compare all the bytes of low-speed keyboard data packets to identify new keypresses in the time between polls.
- There is room for further development of triggers and events using the Pi Pico.

8.2 Future Work

There are many parts of my project that could be explored further, with my work providing a starting point for future developments. The primary area for improvement would be expansion to cover the full bandwidth of a low-speed USB keyboard, when 6 keys are being pressed at once, as this would then produce a complete trigger generation experience for all low speed keyboards (which represents the majority in the market). Once it is capable of handling full low-speed keypresses, it would also be good to expand it to work for other USB speeds as well. Both of these could be achieved by using a microcontroller with a higher clock speed, so it can do the processing in a shorter amount of time between the packets. The Raspberry Pi Pico 2 could be a good starting point to look at for this (as the current code can easily be made to run on it due to the SDK used supporting it). It would be good to expand the trigger generation to work for more types of devices than keyboards, such as mice or even USB 2.0 flash drives, although these would each require higher speed capabilities.

It would also be good to find a way to make the hardware more compact, as it currently is not very space efficient. Finding a way to attach the USB breakout boards directly to the Pico would allow for a closer packing of the components together. This could be achieved with custom components that I did not have the capacity to produce. Finding a lower cost UART to Serial adaptor would also be beneficial, as the Debug Probe represents a significant portion of the cost which can be cut down. Implementing a speaker or LED display would also improve the capabilities of the device by allowing more output events to be triggered. Currently the triggers are limited to only onboard outputs, however there is room for future research to see if it is possible for the device to send its own USB signals down the data lines in response to keystrokes.

Bibliography

[1] R. E. Anderson. Acm code of ethics and professional conduct. *Communications of the ACM*, 35(5):94–99, 1992.

[2] O. Angelopoulou, S. Pourmoafi, A. Jones, and G. Sharma. Killing your device via your usb port. In *Proceedings of the Thirteenth International Symposium on Human Aspects of Information Security & Assurance (HAISA 2019)*, pages 61–72. The Centre for Security, Communications and Network Research (CSCAN), 2019.

[3] ataradov. *usb-sniffer-lite*. `https://github.com/ataradov/usb-sniffer-lite`.

[4] J. Axelson. *USB complete: the developer's guide*. Lakeview research LLC, 2015.

[5] BCS, The Chartered Institute for IT. Code Of Conduct For BCS Members, 2022.

[6] S. Bellini. Accessed via: `https://upload.wikimedia.org/wikipedia/commons/1/1e/USB_Hardware_Keylogger.jpg`.

[7] S. Chahrvin. Keyloggers, pros and cons. *BCS, The Chartered Institute for IT*, 2008.

[8] Computer Misuse Act, 1990. `https://www.legislation.gov.uk/ukpga/1990/18`.

[9] R. Creutzburg. The strange world of keyloggers - an overview, part i. *Electronic Imaging*, 29(6):139–139, 2017.

[10] G. Halfacree and B. Everard. *Get Started with MicroPython on Raspberry Pi Pico: The Official Raspberry Pi Pico Guide*. Raspberry Pi Press, 2021.

[11] M. Komus. Using a raspberry pi pico as a logic analyzer with pulseview. *hackster.io*, 2021.

[12] Microsoft Corporation. Getting started, microsoft windows 98, 1998.

[13] D. V. Pham, A. Syed, and M. N. Halgamuge. Universal serial bus based software attacks and protection solutions. *Digital Investigation*, 7(3):172–184, 2011.

[14] pico-coder. *sigrok-pico*. `https://github.com/pico-coder/sigrok-pico`.

[15] A. Pini. Uarts ensure reliable long-haul industrial communications over rs-232, rs-422, and rs-485 interfaces. *DigiKey*, 2019.

[16] Raspberry Pi Foundation. *Logic Analyser Pi Pico Example*. `https://github.com/raspberrypi/pico-examples/tree/master/pio/logic_analyser`.

[17] C. Recommendation. Definitions of terms concerning data communication over the telephone network, vol. *VIII, Rec*, 7, 1988.

[18] S. Sagiroglu and G. Canbek. Keyloggers: Increasing threats to computer security and privacy. *IEEE Technology and Society Magazine*, 28(3):10–17, 2009.

[19] A. Singh, P. Choudhary, A. k. singh, and D. k. tyagi. Keylogger detection and prevention. *Journal of Physics: Conference Series*, 2007(1):012005, aug 2021.

[20] M. Srivastava, A. Kumari, K. K. Dwivedi, S. Jain, and V. Saxena. Analysis and implementation of novel keylogger technique. In *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*, pages 1–6, 2021.

[21] therealdreg. *Open Hardware Keylogger Implement*. `https://github.com/therealdreg/okhi`.

[22] P. Tuli and P. Sahu. System monitoring and security using keylogger. *International Journal of Computer Science and Mobile Computing*, 2(3):106–111, 2013.

[23] USB Implementers Forum. *Universal Serial Bus 2.0 Specification*, Apr. 2000.

[24] R. Wimmer, A. Schmid, and F. Bockes. On the latency of usb-connected input devices. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–12, 2019.