Hardware

Lea Schirp, Benjamin Stutte, Codruta Gherman, A S M Towfique Hasan

2023

Abstract

Our group was tasked with taking care of the hardware components of the FabArcade, most of all to keep input latency minimal and provide the snappy controls of 80's Arcades. For this, we set up the controller to simulate two discrete controllers via specific libraries to allow for easily adding more or different controllers while keeping the input latency to around 2.5ms through efficient input reading and quick hardware polling. In this document, we describe the setup and properties of this controller. Additionally, we present a design for a simple device to measure different types of latency and discuss its performance.

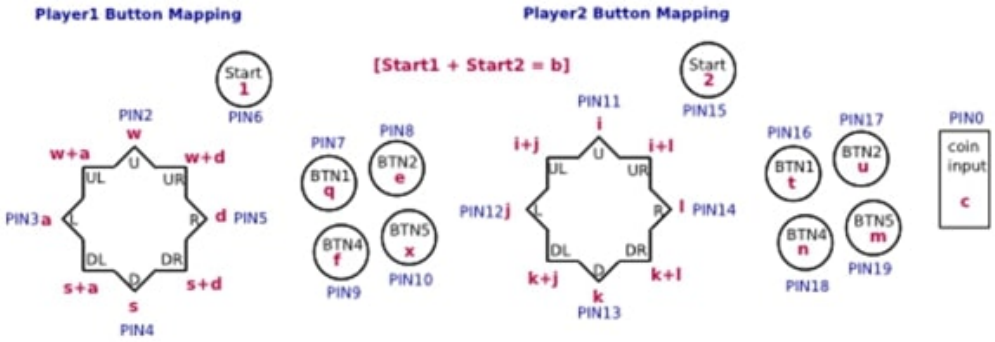

Figure 1: Pin Mapping

1 Introduction

The controller is the central part of the user input for the Fabarcade and with that, has to solve all the central interfacing issues to provide lag-free input. This includes most of all avoiding or minimizing input delays caused by USB jitter and internal processing of the inputs to keep the overall input latency well below 100ms. As this deadline includes processing by the system itself and should not even be perceived as delayed causality, the goal for the controller is to limit delays to latency induced by using a combination of technical methods such as a direct pin placement on an Arduino Leonardo and sending the controller data through the USB connection as an actual joystick. Together with really short debouncing times, all of these measures should deliver a seamless interaction between the user and the Fabarcade.

Our team was assigned to deal with the hardware component of the FabArcade. This includes most of all the controller, dealing with its setup, registration, and latency, as well as finding a way to measure and mitigate latency.

1.1 The Old Controller

The Fabarcade existed in different versions since 2013, however the original setup wasn't running properly anymore, mostly due to issues related with the Raspberry Pi and its SD card. The setup is in detail described in the original documentation[1].

Hardware wise, the old setup consists of an Arduino Uno with a custom-designed shield, which is an interface for all buttons. The joysticks are positioned so they press buttons for every position they can be put in (up, down, left right), while the diagonals of the axes are mapped to the two respective buttons being pressed (e.g. up and left for the upper left position).

In this, the controller was implemented on an Arduino Uno that was put in DFU mode to permanently simulate a keyboard. Each button (including the axes) was mapped to a different key (see fig. 1), so the games would have to work on keyboard-based input alone.

The original shield [2] simply interfaces the buttons with the Arduino. Apart from connecting buttons to pins and a common ground, this shield also hosts common connections for the coin slot and the coin slot override button.

Our first step was to test all parts of the old controller and figure out where to start, leading to the following conclusions:

- Buttons seemed to work fine when testing with a multimeter

- The shield still does what it is designed to do
- The pin connections are followed in the way specified in 1
- Other devices recognize the Controller as a keyboard and accept its input

Considering this, we decided to keep using the old buttons and shield if possible.

2 Setup

The following section describes the setup process and properties of the system. For a more condensed version on how to set everything up etc., please refer to the manual[1] and the original FabArcade Documentation [1] (at least considering the hardware setup, swapping the Arduino Uno with an Arduino Leonardo as mentioned below) instead.

The general process is as follows:

- Gather the described components and build the shield [2]
- Connect the buttons to the shield as described in the pinout1 and the shield's documentation.
- Flash the Arduino Leonardo with the provided controller code. You might need to specifically select the Arduino Leonardo as the used board and add the Arduino Joystick Library to your IDE.
- Connect the Arduino to the FabArcade or whichever device you like to control with it

2.1 Controller

The main components of the hardware setup are:

- Arduino Leonardo
- Arcade joystick
- Arcade button
- FabArcade Shield

Note that this setup will not work with an Arduino Uno. The Leonardo differs from other Arduinos in that it uses a microprocessor with built-in USB communication capabilities and thus doesn't need any special setup to be used as a USB HID device.

2.1.1 Buttons

The following table specifies which Arduino Pins and Ports (in parentheses) a specific button is connected to:

To use the provided code "as is", you need to connect the controls in this way.

Button mappings might not translate the same way into each software framework. In PyGame, which is the framework which all of this year's game groups were using, the mapping translates the following way for each of the players:

As an added benefit, this maps the controller's buttons in the same way an XBox Controller maps its left Joystick and A/B/X/Y controls, with Start mapped to LB and Coin to RB. Games should thus be somewhat compatible to this style of controller.

[1] One sunny day, this footnote will contain a link to the finished manual

Button	Player 1	Player 2
Joystick up	2 (D1)	11 (B7)
Joystick left	3 (D0)	12 (D6)
Joystick down	4 (D4)	13 (C7)
Joystick right	5 (C6)	14 (F7)
Blue	7 (E6)	16 (F5)
Yellow	8 (B4)	17 (F4)
Black	9 (B5)	18 (F1)
Red	10 (B6)	19 (F0)
Start	6 (D7)	15 (F6)
Coin	0 (D2)	

Table 1: Player Pin Mapping

Button	PyGame Control
Joystick Left/Right	Axis 0
Joystick Up/Down	Axis 1
Black	0
Red	1
Blue	2
Yellow	3
Start	4
Coin (P1 only)	5

Table 2: The buttons are mapped in PyGame like this

2.1.2 Software

The Arduino simulates two USB HID devices simultaneously to be able to integrate the controls for both player 1 and player 2 into one Arduino. This is achieved by using the Arduino Joystick Library [4], which is an unofficial add on to the original Arduino Joystick Library with added dynamic HID support.

The Arduino's code reads the ports directly to speed up the processing, as the provided `digitalRead()` functions from the Arduino framework increased latency to intolerable numbers.

After first issues with debouncing where multiple values were sent for each pressed button, an additional delay of 1ms at the end of the loop was added to essentially slow the execution down. What happened here was that between two USB polling intervals, the Arduino went through multiple iterations of its loop and detected quickly changing values, which then were sent all at once. The added delay appears to have resolved this issue (albeit at the cost of a small amount of latency) as no further related issues were raised even during extensive testing conducted by the integration and game groups.

2.2 Raspberry Pi

Considering the games were to be run on a Raspberry Pi 4 with Raspbian, we needed to find a way to improve the USB HID performance of the Raspberry Pi. To improve this and mitigate USB input latency, we added the option `usbhid.jspoll=1` at the end of `/boot/cmdline.txt` to set the polling rate to 1000MHz, which should translate into a lower bound of 1ms for the average input latency.

3 Latency

A major goal of our group's work was the mitigation of overall latency induced by the controller. In practice, this includes two main aspects. First, we had to optimize the system's performance to avoid adding latency, both by the controller itself and by the software running on the Raspberry Pi. The way this was achieved was mostly described in the previous chapters. Second, we needed to find a way to measure latency, which is going to be the focus of this chapter.

Between a button being pressed by the user and the moment the connected action is being displayed on the screen, there are various steps that add latency, some of which can be affected by the controller and the way it is set up, while others are dependent on other factors like game design, display refresh rate, and many more. We thus have to make a differentiation between *end-to-end latency* (the delay between an input and a corresponding output) and *hardware latency* (the delay between an input and the moment the system registers the incoming event). The important part for our setup is the hardware latency, as we have little to no control over the rest of the system. This makes it necessary for us to isolate corresponding input and output events to measure the time between them.

A first approach involved a simple setup for end-to-end latency, where both a button and the screen of the system were filmed with an iPhone 11 running the Is It Snappy? app. This app records high-speed video at 240fps and lets the user step through the video frame by frame to measure the time between events. Measuring latency like this is tedious and not precise, because each measurement takes a considerable time, the precise time when a button is pressed is not unambiguous and the best-case accuracy is around 4ms. Additionally, this method only measures end-to-end latency and does not isolate the latency of the controller alone.

3.1 Latency Setup

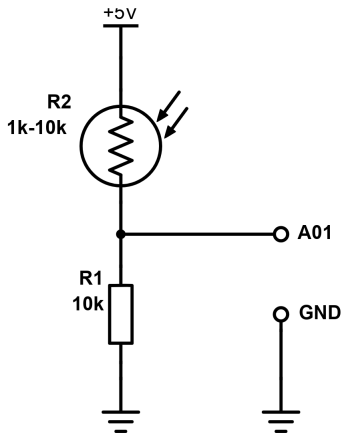

Figure 2: The 1k-10kΩ photo resistor is connected with a 10kΩ pull-down resistor to pin A01

We thus set on a different setup for measuring the input latency as precisely as possible. This setup was inspired by Prof. Borchers' setup to compare the latency of a Sinclair ZX Spectrum against that of a Macbook Pro[2].

[2] https://youtu.be/G3s8t3r9UdA?t=5957

The final version works as follows: An Arduino is hooked up to the controller instead of one of the existing buttons. Additionally, a photo resistor is connected to an input of said Arduino (see Figure 2). It then generates a short pulse to simulate a button being pressed on the controller and starts a timer with microsecond accuracy.

The controller then processes this pulse and at some point sends the corresponding data to the Raspberry Pi. On the Pi, there is a script running that, once a joystick input is registered, flashes an onboard LED, which is placed next to the photo resistor. Once this is registered, the timer is stopped and the resulting time is sent via serial to a laptop, which saves the values.

This setup is not only simple to build, but also system independent, flexible and precise. Events are recorded with a precision in the 10s of microseconds and the setup can easily be modified to measure e.g. end-to-end latency or to wait for existing button events instead of generating its own.

3.2 Standalone

To improve upon the existing setup, we planned and in part implemented a standalone version of the latency setup.

A setup like this should most of all cater to the needs of people who build and test systems with critical latency goals. This leads to the following design requirements:

- Quick, reliable, and precise latency measurements
- Practical application: Users may need slightly different modes in different situations, i.e. a way to test values quickly without attaching a computer
- Modularity: different input devices, measurements, and environments may call for a huge amount of different setups

We thus concluded, that we need a complete system to provide a way to set up different parameters, get immediate feedback on latency, and give users ways to modify this for their setup.

To ensure all of these requirements, we designed a PyBadge[3]-based setup, which provides us with a display, controls, connectors and even a battery for true standalone capability. The general test setup was kept the same - it still generates a pulse and waits for a photo resistor to register an input.

3.3 Limitations

This latency setup, both the primitive and the standalone versions, solve some issues for latency measurements, but they are not without their faults.

Both versions still only have limited precision. While a constant light source of sufficient brightness is registered by the photo resistor within under 20µs, indicating a high precision once the light source is present, the final results only provide an upper bound on latency due to the additional latency that is induced by the feedback mechanism of the script that flashes a light on the Raspberry Pi.

The primitive setup also only starts waiting for the photo resistor to change in value after the full input pulse is sent to the controller. This is solved in a more concurrent fashion on the standalone setup, as this continuously checks the input during the generation of the pulse, which also enables this setup to stay more

[3] https://www.adafruit.com/product/4200

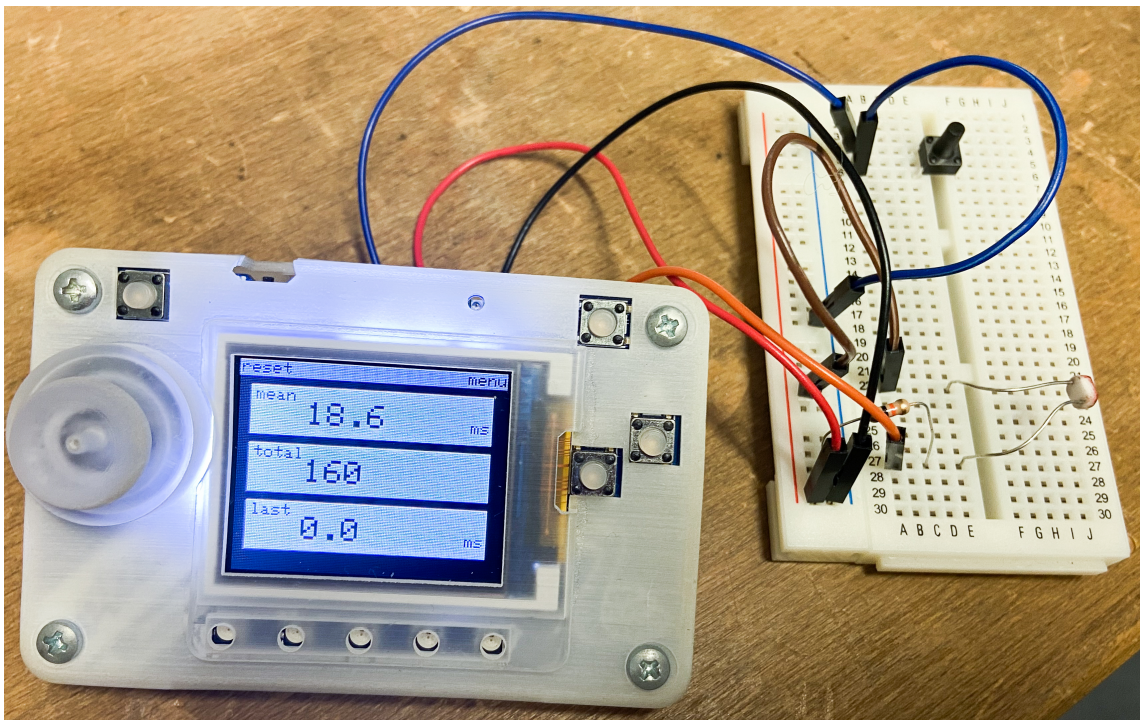

Figure 3: The PyBadge-based standalone latency setup so far. As of now, the circuitry is still located on the breadboard left to the PyBadge.

responsive during measurements. However, ensuring this responsiveness comes with a cost, namely reduced precision; while the primitive setup checks for the value of the photo resistor roughly every 20µs, the standalone setup achieves values around 30µs. This should still suffice for most setups, as this is still well below minimum USB HID polling intervals (1ms) and display refresh rates (usually 120Hz or smaller).

4 Performance

4.1 Method

To evaluate the performance of our controller, we used the primitive latency setup, described in the previous section.

The LED script was started on the Raspberry Pi with the games and the menu installed and the latency setup was connected to the controller like described before. The latency values were recorded with a Python script running on an additional laptop.

We recorded 1000 subsequent latency values, which were then processed using Python.

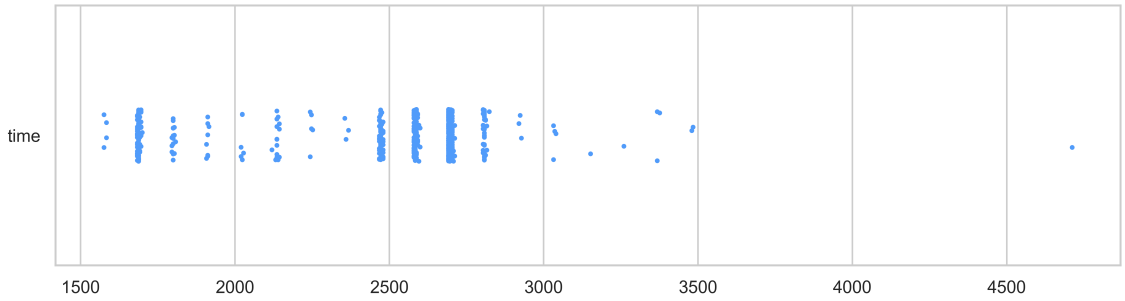

Figure 4: Strip plot of the latency distribution

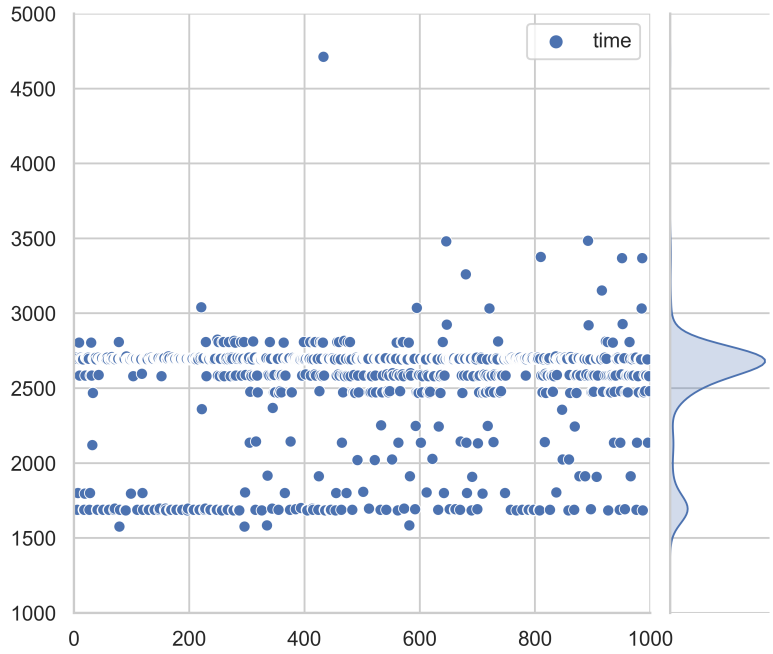

Figure 5: Scatter plot of the latency distribution over time

4.2 Results

The results are displayed in figures 4 and 5.

Latency values are distributed around a mean value of 2524µs with a standard deviation of 356µs and some outliers, most notably one at a maximum of 4712µs. As

apparent in Figure 4, the values are closely clustered around multiple values with no values between two neighboring clusters. The histogram in Figure 5 has two notable peaks, one at around 2600µs and another at around 1600µs.

4.3 Discussion

Considering similar work by Wimmer et al. [3], the latency of our controller is comparable to State-of-the-art input devices, as their study only presents three (out of 36) devices with a similar or faster latency, which is not factoring in that our latency setup adds around 500µs of delay for flashing the LED of the Raspberry Pi.

Out of the 1000 times that the setup simulated a button press, not a single value was dropped, even though input pulses are way shorter than they are in practice. Though responding to such short pulses might create debouncing issues, none of the groups (including us) encountered any issues like that after the delay was added to the controller code (see subsubsection 2.1.2).

The source for the clustering of values couldn't be unambiguously explained within the scope of this work, though its regularity hints at it originating within the Arduino of the latency setup.

5 The Case

Staying with the theme of 1980s-Style gaming, we picked a case for the Raspberry Pi, which resembles the form of a Sinclair ZX Spectrum +3[4]. This was 3D printed with an Ultimaker S5.

References

[1] Anke Brocker and Florian Busch. *Building the FabArcade*. 2013. URL: https://hci.rwth-aachen.de/index.php?option=com_attachments&task=download&id=2108.

[2] *FabArcade Shield*. 2013. URL: https://hci.rwth-aachen.de/fabarcade-shield.

[3] Raphael Wimmer, Andreas Schmid, and Florian Bockes. "On the Latency of USB-Connected Input Devices". In: *CHI '19 Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems Paper No. 420*. New York, NY, USA: ACM, Mai 2019, 420:1–420:12. URL: https://epub.uni-regensburg.de/40182/.

[4] Matthew Heironimus. *Arduino Joystick Library*. https://github.com/MHeironimus/ArduinoJoystickLibrary. 2022.

[4] https://www.thingiverse.com/thing:3736633 Sinclair ZX Spectrum +3 case model on Thingiverse