Design Prototype – Prototroller

Merrick R., Britton M., Caleb O., Evan Z., Yu-yang H.

Effort

To progress the Prototroller to a working prototype state, each team member contributed approximately 40+ hours of work. Of this, around 20% was spent on evidentiary research to determine technical feasibility and hardware/software fitting our design constraints. This identified the RP2040 as the microcontroller of choice for both master and modules, and the Raspberry Pi (RP) Pico as our prototype boards. Around 30% was spent on designing and implementing external interfaces, the modules the user interacts with. The remaining 50% was spent on designing and implementing the internal systems (SPI) along with debugging.

In our pre-alpha build, we attempted and succeeded in implementing a SPI example provided by the RP Pico SDK that implemented master-slave communication. It endlessly cycled a data buffer between devices, demonstrating feasibility of our design as we could build upon this concept and apply multiple chip-selects to communicate with an array of at most 25 devices (by our design). Since the master needed to communicate with more than 1 module, we could not use the specialized SPI chip select. Instead, we had to create our own scheme using the available GPIO pins, i.e., we had to manually control chip selects.

We brainstormed and decided to experiment with button and joystick components to create a prototype that would implement a smaller number of modules (2-4). There was success with implementing SPI communication with a button module. We implemented a separate module consisting of a joystick and an ADC to convert the X/Y analog values to digital data. We used a single GPIO pin to control and read data from one module at a time, and after seeing that the master correctly read data from each module individually, we attempted to combine the two modules to see if we could poll between them.

Attempting to use a common SPI bus was an issue as the Pico does not tri-state the MISO line when its chip select is high (not selected). When a module is selected and sending data to the master over MISO, there will be a “tug of war” situation as non-selected modules attempt to pull MISO low. This resulted in both modules failing to send data, despite only one chip select being driven at a time. A Pico SDK GitHub issue suggested enabling the SPI output enable for MISO as soon as the module is selected. When the module is de-selected, it may disable the SPI output enable so it no longer drives MISO low. This solved our issue, and we were able to connect two modules to our master.

We encountered another issue where the master and modules would be out of sync with data transmission. This was a result of our fix with the MISO line. The master does not wait on the module, instead it is immediately expecting data when it pulls a module chip select. The MISO line fix introduced several cycles on the module until it blocks for the chip select, so we implemented a handshake between the master and selected module with a ~100us sleep on the master. This seemed to fix our synchronization issues. Furthermore, we wanted to know how to identify what module is connected at each chip select, so we added a module identification handshake that sends info about the module itself when the master polls through each chip select. This allows the master to verify input received by the slave and can be used for data corroboration to check if the data is formatted correctly.

To utilize I/O from our modules in a meaningful way, we decided to use USB HID (Human Interface Device) to communicate with a host device (i.e., computer running Windows) to move and click the host mouse using joystick and button modules. We settled on the TinyUSB library since Pico SDK had support

for it. In a separate branch, basic HID code was adapted from an example that allowed a Pico to be detected as an HID-compliant mouse and move the mouse at a static rate.

We then attempted to integrate the HID and SPI subsystems but were met with many issues. Serial logging output was lost once TinyUSB was added to the project—a phenomena that most TinyUSB developers simply tolerate. However, by adding another device descriptor to our TinyUSB code which identifies our device CDC (communication device) in addition to HID, we regained the ability to debug with serial output.

Additionally, the SPI lines appeared to be malfunctioning once more, and the HID device was no longer being recognized. After much speculation, we believed that the print statements used to debug our serial buffers may have taken too much time, and by removing these print statements, the modules would be fully ready to transmit said data whenever required by the master device. Thus, HID and SPI functionality was properly integrated, allowing the mouse to be moved and clicked using the joystick and button modules. We get full functionality even when disconnecting and connecting modules, demonstrating the reconfigurability aspect of the project.

Evidence of Soundness

Theoretical Background

One aspect of the Prototroller yet to be explored is low latency, low power, and reliable operation when many modules, potentially varying in I/O functionality, are connected. SPI communication with a significant number of devices has not formally been explored in terms of qualitative nor quantitative analysis regarding functionality and performance, but theoretical implementation of connecting large amounts of slave devices to a single master device is explained in Piyu Dhaker's *Introduction to SPI interface*. In regular SPI communication, the condition and consequence to adding multiple devices is the increasing number of GPIO pins required to control the chip select. The logic required to pull down specific chip-selects also forbids multiple chip-selects from being pulled high. With these conditions being met, the only restrictions to the number of devices allowed are power resources and responsiveness of slave devices.

Prior Art

Industry implementations of modular controllers have utilized specific slots for components with no ability for repositioning outside of these slots. Often, there are no more than three of these slots on the controller. Nonetheless, it is evident that controller reconfigurability is a possibility upon which we may expand. There also exist devices containing modular components such as MIDI boards. These devices permit module swapping to any location on the grid, but they do not offer a robust HID gamepad format for communication with the host computer, and do not include common gaming controls such as a joystick.

Figure 1: Thrust master ESWAP X Pro Wired Controller

Figure 2 - Mine S Special waves MIDI Controller

Empirical Evidence

We utilized RP Pico boards containing RP2040 microcontrollers to experiment with potential for numerous modules with minimal connections. We have ascertained that twenty-five modules are a viable amount, even at slower baud rates with SPI. Experimenting with a full-duplex SPI baud rate of 1MHz, resulted in a 2ms transmission of 256-byte data packets. This would give a total time of around 51.2 ms for full-duplex communication on all connected modules. Slowly increasing the baud rate, we have seen partial success with a baud rate up to 8 MHz. This indicates we can increase this timing upwards of 80 MHz for ultra-low latency with many active modules, improving responsiveness and thus proving viability of a modular controller where all communication occurs over a bus, rather than dedicated lines.

External Interfaces

Presentation

There are three kinds of modules and additional I/O we prepared that the prototype presents the user with. The joystick is an input module that the user may use to control movement of the cursor on the host, thus serving as a mouse (later – a proper gamepad joystick). There is also a left-click button module and a right-click button module. The user can press the left-click to click, select, drag to highlight a word and/or object on the host. The user can press the right-click button to provide additional information and/or properties of an item selected on the host. A "rescan" button initiates a scan of all module slots to update the internal data store when pressed. When the user connects the host to the master over USB, graphical serial debugging data is also available which gives a view into the system internals. For the final build, we will have optional hand shaped guards that connect to the chassis. This is common in most mainstream controllers as the guards provide a proper

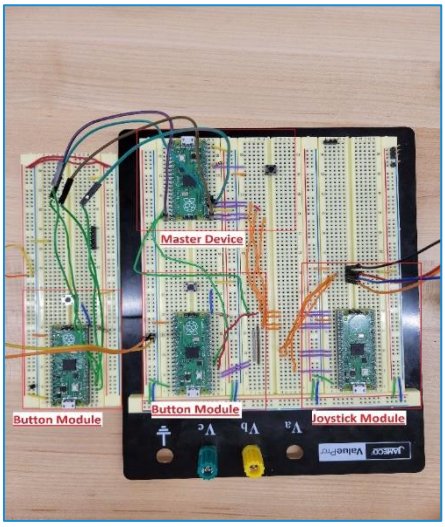

Figure 3: Breadboard Prototype

grip. Further, all guards will contain motors inside them to provide haptic feedback. As they are optional, the user may remove them and place the Prototroller flat.

Perception

Users will perceive a change in state by watching the mouse move according to their inputs. Other modules will also indicate changes in application by reflecting a user input on the connected computer. The state of connected modules can only be perceived by the connection of serial output. If a module is not connected correctly, or a module is put in and the user has not pressed the rescan button, the module will not output anything, as it is in a disconnected state. Additional modules that receive output from the controller and the connected computer will be able to provide output through a graphical interface on the Adafruit Featherwing 128x32 -- potentially displaying a mapping of what modules are currently connected to the board – or other sensory experiences such as haptic tactile feedback or audible responses from a simple buzzer device. All the outputs would be defined by the connected computer or the firmware running on the controller.

Usability

The mouse input and output are predictable and consistent, as well as discoverable. If a user ends up holding the joystick in the wrong configuration (i.e., upside-down), they can very quickly and easily discern the correct orientation through trial and error. Any incorrect or garbled results from the controller are detected and removed before interaction with the interface through our packet verification handshakes. The user can optionally rescan the array of modules if there are incorrect or garbled results from the modules.

Internal Systems

Component Architecture

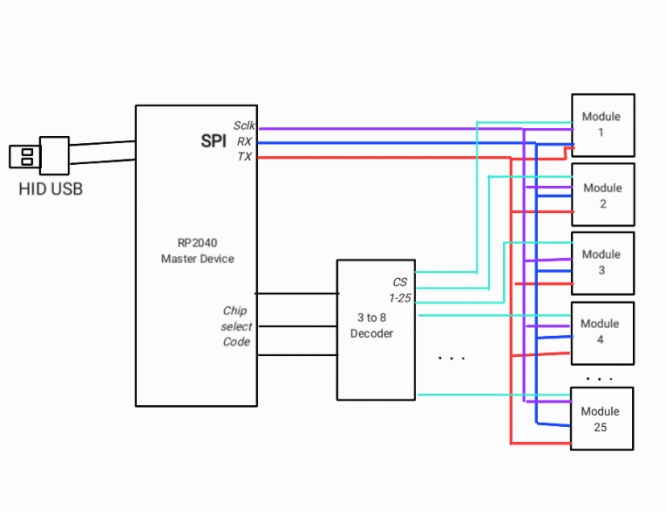

Figure 4: High-level Schematic of Architecture

The project architecture is demonstrated by using a host-master-module communication scheme. The modules that the user interacts with transmit data to the component's dedicated Pico board. This board then forwards that data, when requested, to the master Pico, which then formats and normalizes the data, and sends it to the host device via USB. This fully comprises our project's architecture, and the final artifact will simply elaborate features using this framework.

The firmware is structured with the specific communication behaviors compiled into libraries utilized in tandem with the Pico SDK. A specific RP2040 chip function is written into a source file that is selectively flashed onto the device based on the chip's function.

Communication Mechanisms

Communication between all parts of the project takes place through wires. The communication protocols differ. When the master Pico is communicating with modules, it is using Serial Peripheral Interface (SPI). This is accomplished with the ARM Primecell PL022 located on the RP2040 silicon. This communication occurs when a module is being scanned, or at a period time coded into the firmware for an information update. This communication is expected to always occur, only ceasing when the module is disconnected or has malfunctioned. When the master is communicating with the host computer, it is using USB HID and CDC functionality provided by the TinyUSB library. This communication happens at least once every 10ms but can occur earlier if the controller timings are ahead of schedule. This communication will halt if the host computer rejects the USB connection, but otherwise it occurs continuously while the controller has power.

Resilience

The controller is resilient against improper packet acquisition from connected modules via verification bytes that ensure a module is connected and sending valid data. When this is not true, the module is recorded as temporarily disconnected and any invalid input is not forwarded to the host device. Blocking statements are used within the controller to ensure that data transmissions are synchronous and not in contention when sharing a bus. In the case of device failure, the device disconnects from the USB HID connection until the problem is resolved or the unit is power cycled. Nonetheless, a debounced master rescan button press will perform a manual rescan of all chip selects and serves as our ultimate error correction and recovery mechanism.

We questioned how big input/output buffers should be for communication between master and modules. We defined a packet as 256 bytes, and the master will expect 256 bytes from every module. We encountered the problem that data is missing between module polls and potential waste of latency using a 256-byte packet for minimal-data components (ex. button). Moreover, we were concerned with the case that the module reads/writes more than 256 bytes - bigger than the packet size. We were assuming the module can tell the master how many packets it can expect to receive as part of the identification handshake, and subsequently make buffers more “dynamic.” However, we decided to focus on more pressing matters and delegate this as a Design 2 advanced firmware task.

Building & Running

Instructions to build the project are located on our design repository at roboryman/prototroller (github.com). The build process produces software artifacts (binary .uf2 images) for the master microcontroller and all module variants for the slave microcontrollers. Our project is heavily based around a hardware artifact, the physical Prototroller interface. You must have this hardware to flash the software artifacts. In our prototype state, the physical Prototroller interface is an array of breadboards with RP Pico’s acting as the master and modules.

To build a working master breadboard, one must define the number of GPIO pins desired, and if over 3 utilize a hardware decoder to implement chip select signals. The SPI pins defined must be connected to all module breadboards on a common bus. A form of button must also be connected to the rescan pin to enable module detection.

To build a working module breadboard, one must connect the Pico SPI0 pins denoted in the module source file to the pins noted in the master source file. The desired module hardware must also be wired to the defined GPIO pins or ADC pins. Once the Pico is supplied with power, these Pico modules will be capable of communicating with the master.

A demo video showcasing the overall prototype including the hardware setup is available here: https://www.youtube.com/watch?v=UJIYk-D0DHo