Gaming Handheld Audio Controller

Silas Wang

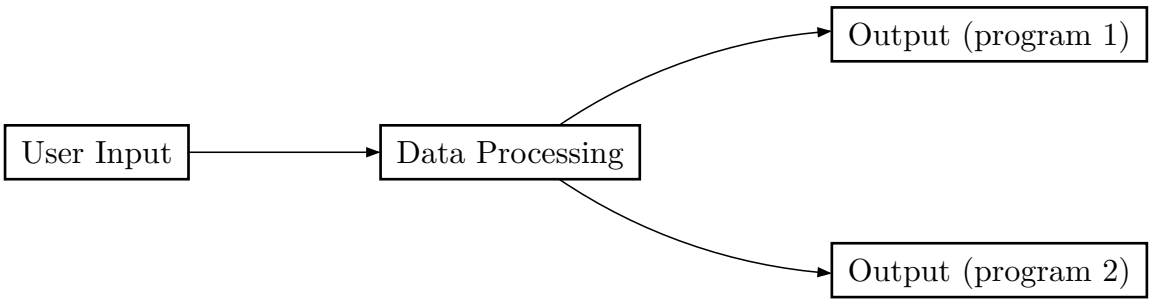
Electronic Product Design Fall 2025

Contents

1. Overview

At a high level, my Electronic Product Design prototype is a device that is capable of controlling both a video game and audio equipment (be it, music hardware or software).

You can technically write this device off as a MIDI controller, but this device has the ability to implement any kind of communication protocol
(CV, MIDI, OSC, HID, etc.) and simulatneously control two devices that each require a different protocol to communicate with it. You can think of it as the following flowchart:



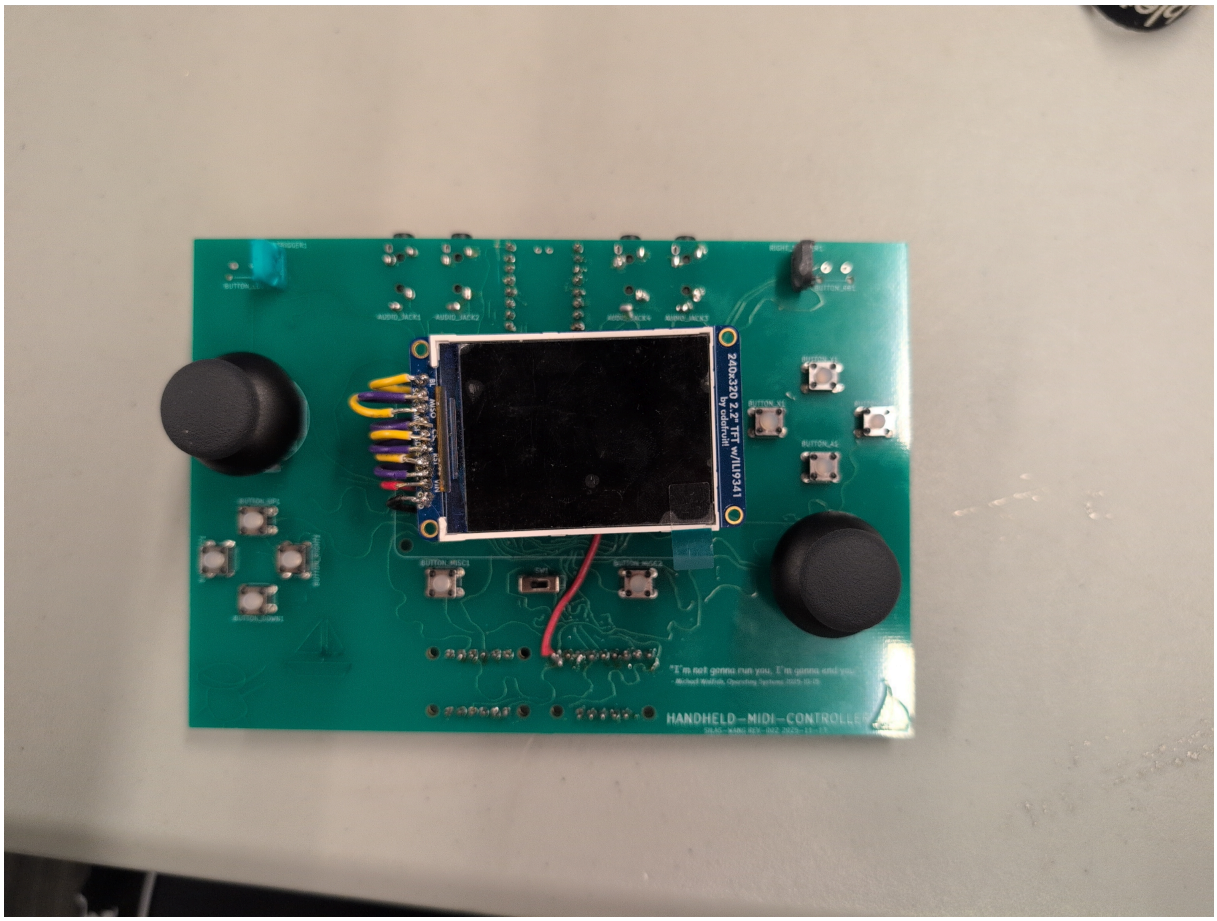
where *User Input* outlines the User interacting with the device and its parameters, *Data Processing* refers to the firmware witin the device, and *Output* refers to the different devices (software or hardware) that it can control. The current version of this prototype focuses on creating a device that can communicate with MIDI devices and video games.

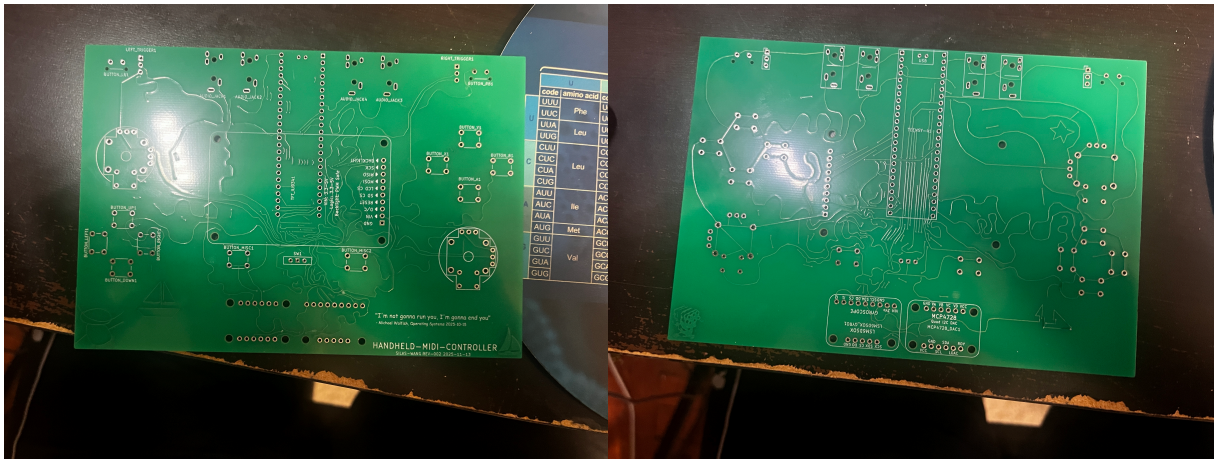
Future designs will align with the Y2K transparent-tech style heavily inspired by tech companies like Nothing, and technology from the Y2K era. See Section 4 for more information.

Handheld gaming controllers are popular choices for gaming due to their form factor and convenience. Their integration with existing IP and the combination of hardware and software optimization lead to a device powerful enough to run demanding game titles whilst being light enough to carry around in a backpack. Some examples of these include the Steam Deck, Nintendo Switch, and Lenovo Legion GO.

To my knowledge, there is little exploration for products that balance the "sound design by playing video games" mantra. This prototype and the subsequent documentation attempts to address this gap by thoroughly documenting what I have learned whilst creating this project.

An image of the prototype and PCB can be found below:





A video demonstration can be found here.
All code (test `.ino` files, classes, etc.) can be found on GitHub.
All parts and software used can be found at Section 6

2. User Interface

The parameters on the user interface is modeled after gaming controllers such as the Xbox controller or the PlayStation controller. Two joysticks, each with a select button are utilized alongside two four-button arrays, one labeled DPAD and the other XYAB respectively. In addition, the prototype board has the capacity to support two trigger buttons, two analog triggers, and two "miscellaneous" buttons, akin to a typical gaming controller.

Higher end gaming controllers utilize Hall-Effect Sensors, which are said to offer higher precision, finer deadzones[1], and a longer product lifespan. For the sake of this documentation, we assume no differences.

Additionally, a few extra components are used. A gyroscope is used to track the position and orientation of the prototype to offer a more expressive set of gestures that can be translated as MIDI, and a DAC is utilized to write the digitized values collected from sensors as voltage, so hardware music devices can also be controlled. A 2.2″ TFT ILI9341 display has also been used to provide a visual interface for the firmware. The exact gyroscope and DAC I got from Adafruit is the LSM6DSOX Gyroscope and MCP4728 DAC. Additionally, a switch is used to bypass the Keyboard and Mouse, making it easy to set up MIDI CC parameters without destroying a DAW project file. General usage would be

1. load a project file, DAW supporting MIDI-CC of choice
2. enable bypass switch
3. map parameters and hit record
4. open desired video game
5. play the video game

This project utilizes the Teensy 4.1 microcontroller. At the project's current state, values are read into sensors via `analogRead()` or `digitalRead()` function calls, smoothed via a simple moving average filter to remove potential analog noise, then translated into MIDI and HID data. In essence, we configure our Teensy microcontroller to behave as both a MIDI controller and a keyboard/mouse pair.

Future iterations of the project will incoroporate some sort of data processing software and a GUI for the display.

For more information on the design plans for the fabrication of the device, see Section 4.

The PCB was designed in KiCad, inspired by Mason Mann, and printed by JLPCB.

[1] analog joysticks tend to drift / produce noisy values by nature of hardware and its interaction with the world around it. This makes it difficult for firmware programmers to accurately determine the minimum and maximum values of this sensor without setting a *deadzone*, a small range around the joystick that is treated as its resting state.

3. Challenges

Just like the many amazing peers who have taken Product Design with me in the Fall of 2025, this project had a plethora of challenges that needed workarounds for the product to get to its current iteration. This is thoroughly covered below:

3.1. Composite USB Devices

This was one of the core components of my project and arguably the sole reason why this project works in the first place. The Arduino IDE allows users to define a "USB Mode" under the `Tools > USB Mode` tab. For the Teensy 4.1, a plethora of USB Modes are offered, ranging from Serial to 16-Channel MIDI to flight simulators. However, for the goals that I wished to accomplish with the Teensy, there was no option within this dropdown that allowed me to send USB, MIDI, and HID (Keyboard and Mouse) at the same time.

For context, computers send messages that are solely comprised of zeros and ones. This protocol is what determines the means in which these binary strings are interpreted. After all, the main difference between MIDI and other digital communication protocols lies within how different strings are interpreted[2].

USB is inherently a Serial architecture. It sends information bit by bit in series. When a computer powers on, a process within the Operating System runs a routine that labels all connected devices and assigns each device a respective address through a process called enumeration. One of the steps within this process is determining which type of data the device wishes to transfer:

- *Interrupts* typically come from mice or keyboards and don't send a lot of information
- *Bulk* typically comes from printers, who receive large amounts of information at a time and requires extra verification to check if the information is correct.
- *Isochronous* typically involves sending real time data and no error correction (audio)

During enumeration, host machines like your computer look for USB Descriptors within each USB device that is connected to the USB bus of a computer / USB accepting device. Within these USB Descriptors lies all the information that the host needs to properly communicate with the devices. So if we want our computer to think our Teensy 4.1 is simultaneously a computer keyboard, a mouse, and a MIDI controller at the same time, we must look to adjust the Teensy's usb descriptors. We can accomplish this by doing the following:

[2] as well as how the messages are formatted

3.1.1. Steps

1. Open Finder and navigate to `Users/Library/Arduino15`. There are a couple of ways to find this folder, as it is hidden to users by default.
   1. `Command + Shift + G` and copy-paste `~/Library/Arduino15`
   2. `Command + Shift + .` whilst in your user folder (`Macintosh HD/Users/username-here`)
2. Navigate to the following folder: `packages/teensy/hardware/avr/1.59.0`. The entire filepath should then be `~/Library/Arduino15/packages/teensy/hardware/avr/1.59.0`.
3. Open `boards.txt` and `cores/teensy4/usb_desc.h` using your favorite text editor. **we are halfway done**.
4. We need to create a new menu interface for this USB device to show up in.
   - In `boards.txt`, we must add the following lines, replacing `customusbserial` with the name of the new USB Mode:

```
teensy41.menu.usb.customusbserial = Custom USB Serial
teensy41.menu.usb.customusbserial.build.usbtype = USB_CUSTOMUSBSERIAL
teensy41.menu.usb.customusbserial.upload_port.usbtype = USB_CUSTOMUSBSERIAL
teensy41.menu.usb.customusbserial.fake_serial = teensy_gateway
```

This project uses the Serial monitor (for debugging), MIDI, and HID, so I added the following lines:

```
teensy41.menu.usb.serialmidihid = Serial + MIDI + HID
teensy41.menu.usb.serialmidihid.build.usbtype = USB_SERIAL_MIDI_HID
teensy41.menu.usb.serialmidihid.upload_port.usbtype = USB_SERIAL_MIDI_HID
teensy41.menu.usb.serialmidihid.fake_serial = teensy_gateway
```

After doing so, we should see the USB Mode appear in the `Tools > USB Mode` dropdown.

5. We need to link the preexisting USB Descriptors to the USB mode we made for the Arduino IDE.
   - In `usb_desc.h`, we must add a USB Descriptor configuration that matches our desired specifications. Notice that each USB Mode has their USB Descriptors to be defined here. In particular, each USB Mode begins with a `#elif defined(USB_CUSTOMUSBSERIAL)` and features a lot of `#define` lines. You may also have noticed that there are a lot of comments within the beginning of the file that explain a lot of what I will be writing in more depth. You can and should reference that as well.
   - Find a pre-defined USB Mode that is close enough to the specifications that your desired USB Mode is trying to accomplish. Copy-paste and rename it to whatever you renamed the `.build.usbtype` from the previous step.
   - Find other pre-defined USB specifications that match what you are missing.

For this project, I accomplished this by creating an `#elif defined(USB_SERIAL_MIDI_HID)` and copying the USB Serial + MIDI USB Mode to start out with. I then added parts from the `USB_SERIAL_HID` configuration to ensure

that my Serial + MIDI + HID USB Mode has the adequate HID descriptors for the custom USB Mode to properly function. That leaves me with the example below.

6. Endpoints.
    - So far, our picture of how the Teensy communicates with a host has been pretty clear, excpet for one crucial part: How exactly does the Teensy and the host know where leave this information such that the other party picks it up? The answer is Endpoints. Endpoints can be thought of as a rendezvous that both the host and Teensy are aware of. A location where the Teensy knows will reach the host, and visa versa. The Teensy has 8 bidirectional endpoints, indexed at zero (0-7). Since each endpoint is bidirectional, each endpoint can be used for both IN / TX (Teensy -> Host) and OUT / RX (Host -> Teensy) definitions. This means the Teensy has 16 (8 in, 8 out) total endpoints.
    - The first two endpoints (0 and 1) are reserved. You cannot use them. Endpoint 0 is used to send and receive information regarding USB enumeration. Endpoint 1 is reserved for the USB port, leaving us the users 12 (6 IN / TX, 6 OUT / RX) total endpoints for us to use.
    - There are some caveats we must abide by:
        - Each endpoint can simultaneously send and receive, meaning it is possible for us to `#define MIDI_RX_ENDPOINT 4` and `#define MIDI_TX_ENDPOINT 4` and not have an unhappy Teensy. The catch is that only one endpoint can be used for each function that sends data to the host. If the HID Keyboard is sending data at endpoint 5, there cannot be a MIDI controller using endpoint 5 as well.
        - Some functions require more endpoints than others. Consult the elements in which you copied to find out how many TX and RX endpoints are needed per function.
        - You can configure how each endpoint will behave by using `#define ENDPOINTN_CONFIG` to `ENDPOINT_RECEIVED_BULK`, `INTERRUPT, or UNUSED.` You can determine which to use based on the materials you copied earlier.

3.1.2. Final `boards.txt` Modification

```
#elif defined(USB_SERIAL_MIDI_HID)
  #define VENDOR_ID 0x16C0  // you probably do not need to rename these.
  #define PRODUCT_ID 0x488
  #define MANUFACTURER_NAME {'M', 'i', 's', 's', 'i', 'l', 'e', ' ',
                             'S', 'i', 'l', 'o'}
  #define MANUFACTURER_NAME_LEN 12
  #define PRODUCT_NAME {'S', 'e', 'r', 'i', 'a', 'l', ' ', '&', ' ',
                        'M', 'I', 'D', 'I', ' ', '&', ' ', 'H', 'I', 'D'}
  #define PRODUCT_NAME_LEN 18
  //FROM MIDI & SERIAL
  #define EP0_SIZE    64
  #define NUM_ENDPOINTS         7
  #define NUM_INTERFACE     6 // (serial = 2) + (hid = 3) + (midi = 1)

  //ENDPOINTS
```

```
  #define CDC_ACM_ENDPOINT 2 //Serial
  #define CDC_RX_ENDPOINT 3
  #define CDC_TX_ENDPOINT 3
  #define MIDI_RX_ENDPOINT 4
  #define MIDI_TX_ENDPOINT 4
  #define KEYBOARD_ENDPOINT 5
  #define MOUSE_ENDPOINT 6
  #define JOYSTICK_ENDPOINT 7

  #define CDC_IAD_DESCRIPTOR  1
  #define CDC_STATUS_INTERFACE  0
  #define CDC_DATA_INTERFACE  1 // Serial
  #define CDC_ACM_SIZE          16
  #define CDC_RX_SIZE_480       512
  #define CDC_TX_SIZE_480       512
  #define CDC_RX_SIZE_12        64
  #define CDC_TX_SIZE_12        64

  #define MIDI_INTERFACE        2
  #define MIDI_NUM_CABLES       16
  #define MIDI_TX_SIZE_12       64
  #define MIDI_TX_SIZE_480      512
  #define MIDI_RX_SIZE_12       64
  #define MIDI_RX_SIZE_480      512

  #define MOUSE_INTERFACE       3 // Mouse
  #define MOUSE_SIZE            8
  #define MOUSE_INTERVAL        1

  #define KEYBOARD_INTERFACE    4 // Keyboard (Media Keys apparently are a
separate thing.)
  #define KEYBOARD_SIZE         8
  #define KEYBOARD_INTERVAL     4

  #define JOYSTICK_INTERFACE    5 // Joystick
  #define JOYSTICK_SIZE         12  //  12 = normal, 64 = extreme joystick
  #define JOYSTICK_INTERVAL     2

  #define ENDPOINT2_CONFIG ENDPOINT_RECEIVE_UNUSED +
ENDPOINT_TRANSMIT_INTERRUPT
  #define ENDPOINT3_CONFIG ENDPOINT_RECEIVE_BULK + ENDPOINT_TRANSMIT_BULK
  #define ENDPOINT4_CONFIG ENDPOINT_RECEIVE_BULK + ENDPOINT_TRANSMIT_BULK
  #define ENDPOINT5_CONFIG ENDPOINT_RECEIVE_UNUSED +
ENDPOINT_TRANSMIT_INTERRUPT
  #define ENDPOINT6_CONFIG ENDPOINT_RECEIVE_UNUSED +
ENDPOINT_TRANSMIT_INTERRUPT
  #define ENDPOINT7_CONFIG ENDPOINT_RECEIVE_UNUSED +
ENDPOINT_TRANSMIT_INTERRUPT
```

3.1.3. Alternative: `tinyUSB`

Alternatively, we can utilize TinyUSB to do all this work for us. TinyUSB is a USB device that allows for the creation of composite USB devices without the need to edit internal Teensyduino system files. Adafruit provides a nice abstraction with their Adafruit_TinyUSB library that can easily create a MIDI and HID and Serial device. A simple example is listed below for creating a Serial and MIDI composite device:

```
#include <Adafruit_TinyUSB.h>
Adafruit_USBD_MIDI usbMidi;
Adafruit_USBD_Serial usbSerial;

void setup() {
  // begin the device if we haven't yet.
  if(!TinyUSBDevice.isInitialized())
    TinyUSBDevice.begin(0);
  usbSerial.begin(115200); // begin the serial device, other bookkeeping
stuff
  usbMidi.setStringDescriptor("TinyUSB MIDI");
  MIDI_CREATE_INSTANCE(Adafruit_USBD_MIDI, usbMidi, MIDI);
  MIDI.begin(MIDI_CHANNEL_OMNI);

  // failsafe-if already enumerated, additional class
  // driver begin() for stuff like msc, hid, midi won't
  // kick in until re-enumeration.
  if(!TinyUSBDevice.mounted()) {
    TinyUSBDevice.detatch();
    delay(10);
    TinyUSBDevice.attach();
  }

  // don't exit setup() if we're not set up
  while(!TinyUSBDevice.mounted()) delay(10);

  // the rest of your setup() stuff goes here
}

void loop() {
  #ifdef TINYUSB_NEED_POLLING_TASK
    TinyUSBDevice.task();
  #endif

  // failsafe
  if (!TinyUSBDevice.mounted()) return;

  // the rest of your loop() stuff goes here
}
```

3.2. Passing Function Arguments

Computer Science programs like the one at the Courant School stress ideas such as abstraction, the idea of consolidating routines into functions that can be called elswehere with one line of code. C++ has the ability to directly manipulate the different types of memory on the Teensy that is readily available, a feature native to the C programming language[3].

As programs grow in complexity, it's common to abstract data and functions into Objects for readability. But in doing so, it's important to understand the following distinction between passing by value, reference, or pointer. The following table outlines the difference without getting into the weeds of memory management, pointers, or compunter architecture.

	Pass by Value	**Pass by Reference**	**Pass by Pointer**
Function	`read(BUTTON b)`	`read(BUTTON& b)`	`read(BUTTON* b)`
Usage	`read(b)`	`read(b)`	`read(&b)`
Notes	• accessing: `obj.valueOrFunction` • copies object into function, values stored into said copy. copy gets deleted at the end of the function • **doesn't save your data**	• accessing: `obj.valueOrFunction` • gives function an "alias" of the object, modifies actual object • must be initialized • cannot be null, cannot be changed/reseated • **saves your data**	• accessing: `obj->valueOrFunction` • gives function the memory address of the object, modifies actual object • doesn't need to be initialized • can be null and changed/reseated • **saves your data**

A simple example program has been provided below for context for the error I designed by accident. A `BUTTON` struct is defined to encapsulate the data required to get a sensor to properly talk to the microcontroller. Three buttons are declared and initialized, and three `readButton` functions are defined, each using an argument passing method mentioned above. The example is shown on the next page.

Based on the table above, `readButton1` will not change, but `readButton2` and `readButton3` will, because `readButton1` will store any changes made to the object to the local copy stored in the function. So by the time the function exits, the local copy will get removed and no read data will be saved. `readButton2` and `readButton3` doesn't accomplish this by sending some form of pointer to the function, effectively directing the function to save its values at the original object that has been declared within a given program.[4]

[3]This may be common sense to some people studying Computer Science. I write this section because A) I did not pay attention during the C lectures and B) for anybody else in the back who is interested

[4]This is here solely for reference, this is not a lesson in memory management. You can find information about pointers and references on YouTube or in a Computer Architecture class

3.2.1. Example

```
// Arduino program comparing pass
// by value, reference, & pointer.

// Struct that aggregates all
// potentially meaningful
// variables pertaining to the
// buttons we wish to use
struct BUTTON {
  int pin;
  bool prev_state;
  bool state;
  String name;
}

// define global buttons
struct BUTTON button1 = {3, false,
  false, "Button 1"};
struct BUTTON button2 = {4, false,
  false, "Button 2"};
struct BUTTON button3 = {5, false,
  false, "Button 3"};

void setup() {
  pinMode(button1.pin, INPUT_PULLUP);
  pinMode(button2.pin, INPUT_PULLUP);
  pinMode(button3.pin, INPUT_PULLUP);
}

void loop() {
  readButton1(button1);
  readButton2(&button2);
  readButton3(button3);
  Serial.println("====\nButton 1:");
  Serial.println(button1.state);
  Serial.println(button1.prev_state);
  Serial.println("====\nButton 2:")
  Serial.println(button2.state);
  Serial.println(button2.prev_state);
  Serial.println("====\nButton 3:");
  Serial.println(button3.state);
  Serial.println(button3.prev_state);
}
```

```
// read, store, and print data;
// FAULTY as it is pass by value
void readButton1(BUTTON b) {
  b.state = !digitalRead(b.pin);
  if(!b.prev_state && b.state)
    Serial.println("1");
  b.prev_state = b.state;
}

// read, store, and print data
// (but it actually works)
void readButton2(BUTTON* b) {
  b->state = !digitalRead(b->pin);
  if(!b->prev_state && b->state)
    Serial.println("2");
  b->prev_state = b->state;
}

// read, store, and print data;
// (but it actually works)
void readButton3(BUTTON& b) {
  b.state = !digitalRead(b.pin);
  if(!b.prev_state && b.state)
    Serial.println("3");
  b.prev_state = b.state;
}

/*
* Remember, there is no difference
* functionality wise between
* readButton2 and readButton3
* for this given context. There
* are characteristics listed in the
* table which may actually make a
* big difference in some cases, but
* for reading and storing button
* data, there is no difference.
*
* Though, some people prefer pass
* by reference so you don't have to
* write a "->" in place of the "."
* for every method call.
*/
```

3.3. Display Optimization

In principle, a display like the ILI-9341 TFT can give the illusion of a moving image by simply displaying a thing, removing that thing, and very quickly displaying another thing. Imagine that the thing we wanted to display is a variable that changes as a parameter or sensor changes over time. Naively, we can implement something akin to the following pseudocode:

```
void loop() {
  print the value on the screen
  clear the screen
}
```

Depending on how `clear the screen` is implemented, this can be very inefficient. The small SD1306 displays that Steve probably has you use will struggle in performance if they were given an implementation of the above, since the `clearDisplay()` function in the `Adafruit_SSD1306` library will iteratively set every single pixel to black on the screen. This is especially important for dealing with larger multicolor displays. We can address this by modifying the pseudocode above to only clear the values that are changing:

```
void loop() {
  if(a value has changed) {
    print the value to the screen
    clear only the value that has changed
  }
}
```

3.3.1. Implementation

An implementation that I incorporated to this project using the `Adafruit_GFX` library and the Adafruit ILI-9341 TFT Display involves two functions: `display()` and `clear()`. The code below assumes we have the following structures, but you do not need to organize your code in such a manner.

```
struct BUTTON {
  int pin;
  String name;
  bool prev_state;
  bool state;
};
```

```
struct POT {
  int pin;
  String name;
  int prev_val;
  int val;
};
```

```
void display(Adafruit_ILI9341& display, String value, int x, int y,
             int fColor, int bColor) {
  display.setCursor(x, y);
  display.setTextColor(fColor, bColor);
  display.print(value);
}
void clear(Adafruit_ILI9341& display, String value, int x, int y) {
  display.setCursor(x, y);
  display.setTextColor(ILI9341_BLACK, ILI9341_BLACK);
  display.print(0xDA); // a black square, clears one character
}
```

This lets us write the following functions that let us print Strings with a fast refresh rate to the display:

```
void printButton(Adafruit_ILI9341& display, BUTTON& b, int x, int y,
                 int fColor, int bColor, int debug = true) {
  if(b.prev_state != b.state) { // print only when the value is different
    String toPrint = b.name + ": " + b.value;
    clear(display, toPrint, x, y);
    display(display, toPrint, x, y, fColor, bColor);
  }
  if(debug) {
    Serial.println(b.name);
    Serial.println(b.state);
  }
}

void printPots(Adafruit_ILI9341& display, POT& p, int x, int y,
               int fColor, int bColor, int debug = true) {
  if(p.prev_val != p.val) { // print only when the value is different
    String toPrint = p.name + ": " + p.value;
    clear(display, toPrint, x, y);
    display(display, toPrint, x, y, fColor, bColor);
  }
  if(debug) {
    Serial.println(p.name);
    Serial.println(p.value);
  }
}
```

In a `loop()` function:

```
BUTTON b = {3, "Button", false, false}; // pin, name, prev state, state
Adafruit_ILI9341 display;

void setup() {
  pinMode(b.pin, INPUT_PULLUP);
  display.begin();
}

void loop() {
  readButton(b); // we assume this reads and stores a button's value
  printButton(display, b, 100, 100, ILI9341_RED, ILI9341_BLACK);
}
```

3.3.2. Using `millis()`

Another way you can implement a high refresh rate is by using `millis()`, where instead of relying on the internal clock speed of the Teensy's CPU, we can standardize the rate at which we clear and print to the screen based on a predefined frame rate:

```
#define FRAME_RATE 30
unsigned long lastFrame = 0;
```

```
BUTTON b = {3, "Button", false, false}; // pin, name, prev state, state

void loop() {
  readButton(b); // digitalRead() + storing in button struct
  if(millis() > lastFrame + FRAME_RATE) {
    lastFrame = millis();
    printButton(b, 100, 100, ILI9341_RED, ILI9341_BLACK);
  }
}
```

However I have found that with a lot of information on the screen (displaying rapidly changing values) all methods mentioned above exhibit performance issues. A potential solution is highlighted below for anybody who needs to efficiently print 12+ rapidly changing parameters in real time onto a display.

3.3.3. Framebuffers

Future iterations of the project plan on implementing a frame buffer[5]. Frame buffers acknowledge the fact that there is both a CPU and time cost that must be paid when rendering an image onto a display. So they stockpile pixel information inside of an array (or some other data structure) to pay this CPU and time cost once in bulk. Typically two or three frame buffers are used in tandem so when one frame buffer is displaying information, we can write pixels to another frame buffer and hide the bulk CPU and time cost behind an image that is already being displayed. If you wish to implement a frame buffer on your own, the pseudocode is outlined below:

```
framebuffer()
  input: 2 arrays / framebuffers, T time units, information to write
  goal: print to screen efficiently.

  repeat:
    1. write information to framebuffer (array)
    2. wait for hardware to draw framebuffer
    3. swap and repeat steps 1 & 2 with the other framebuffer
```

Frame buffers are incredibly efficient and are the backbone to modern display drivers that regularly appear in flatscreen televisions, high-refresh rate gaming monitors, and touchscreens for mobile devices. As it turns out, we can also optimize this even further with differential updates. Instead of uploading a display-sized array to the display driver or graphics card, we only upload the pixels that have changed; ultimately reducing the memory overhead on a computationally intensive task for a microcontroller.

[5]This is also because Adafruit's `Adafruit_ILI9341` display firmware does not come with a framebuffer built-in. It probably does not matter, but it is an optimization I'm interested and excited about because it allows for cooler stuff to be displayed at a smaller cost.

3.4. Object Oriented Design

Note that this following section has not been implemented in the source code on GitHub. Additionally, the system I propose in this section requires some C++ programming context (which will be outlined in the following section.

3.4.1. Overview

The beauty and detriment that of C++ and Object Oriented Programming lies within objects and cryptic error messages. The software side of this project involves a data processing environment that aims to allow the user to process and control the data in interesting ways such that the output can deviate from raw sensor data.

C++ is an object oriented programming language, which means the user can create their own datatypes that contain functions and variables that are distinct to said datatype. An interesting property of Object Oriented Programming is the principle of inheritance[6], where we can create datatypes using a previous datatype. For example, consider the `Scale` class below:

```
class scale {
public:
  scale(std::vector<int> notes);
  ~scale();
  virtual void arpeggio(); // because it is virtual, we must implement it.
  int play(int scale_degree);
  int playMelody(int melody[], int rhythms[]);
  void transpose(int i);
protected: // this isn't private for a reason discussed later
  std::vector<int> notes;
};
```

It's an incredibly rudimentary scale class that stores some MIDI notes defined by the user and plays a given scale degree. However, if we wish to create a `CMajor` or `DMajor`, we can use this `scale` class as a framework so we do not need to write each function and variable again.

```
class cMajor : public scale
{
public:
  cMajor() : scale({notes to cmajor}) {}
  ~cMajor();
  // we write override to let C++ know that this is the version of the
  // function to use. Otherwise, C++ would have to pick between the
  // parent object (scale)'s methods or the child object (cMajor)'s
  // methods to use.
  void arpeggio() override; // don't need override here, but compiler likes it
  int play(int scale_degree) override;
  int playMelody(int melody[], int rhythms[]) override;
  void transpose(int i) override;
};
```

[6] see this link for more information

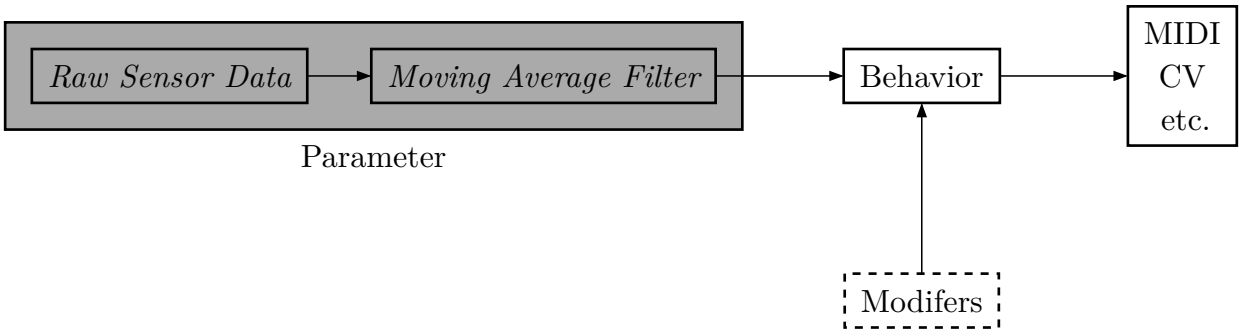
This principle is used quite a bit for the example I propose below.

3.4.2. Design

My Product Design system roughly works in the following flowchart:

Although I plan on adding extra features (see Planned Features), for simplicity's sake, we can assume the following system for the "Data Processing":

Each parameter that the Teensy can read values to store in a variable must have a **behavior**, a set of rules that the parameter will follow. If we zoom into the Data Processing field from the flowchart above, this would give us the following:



3.4.2.1. Parameter Class

Note that when each parameter gets read by the Teensy, it also gets smoothed to remove any analog noise. This is why the "Moving Average Filter" step exists. You can think of the italicized boxes as apart of a "Parameter" that the user can interact with. In code, this means we can view each parameter as the following:

```
// quickly abstracting pins for convenience:
typedef struct device_pin {
  String name;
  int pin;
} device_pin;

// enums for easily reading and labeling parameters
enum class PARAM_TYPE {
  POT,
  BUTTON,
  GYRO,
  MISC
};

// mostly abstract base class. This is similar to Interfaces in Java.
// That is also why we do not have a constructor or destructor here.
class base_param
{
  virtual bool begin(); // intialize parameter
  virtual void read(); // read raw value
  virtual void debug(); // enable serial printing
```

```
  virtual void process(); // smooth raw value
};

// base parameter class of typename N
template <typename N>
class param : public base_param
{
public:
  param() = default;
  param(String n, struct device_pin p) : pin(p), name(n) {}
  param();

  bool begin() override;
  void read() override;
  void process() override;
  void debug(bool s, bool d) override;

  // getters and setters
  N get_val();
  N get_raw_val();
  device_pin* get_device_pin();
  String get_pin_name();
  String get_name();
  void set_name(String n);
  void set_pin_name(String n);
  void set_pin(int p);
  void set_pin_struct(device_pin p);

protected:
  String name;
  device_pin pin;
  N processed_value; // after smoothing
  N prev_raw_val; // previous value
  N raw_val; // raw value
};

// example use with a button
class button : public param<bool>
{
public:
  button() : param<bool>("untitled button", {"nonexistent pin", -1}) {}
  button(int pin, String pin_name, String param_name) :
    param<bool>(param_name, {pin_name, pin}) {}
  bool begin() override;
  void debug(bool serial, bool display) override;
  void read() override;
  void process() override;
  PARAM_TYPE get_param_type() {
    return type;
  }
private:
```

```
  PARAM_TYPE type;
};
```

3.4.2.2. Behavior Class

Because behaviors can potentially get quite involved, I we abstract their logic into a separate object. So then we can imagine a setup like the following:

```
// mostly abstract base class.
template <typename IN, typename OUT>
class base_behavior
{
public:
  base_behavior() = default;
  ~base_behavior() = default;

  virtual OUT process(const IN& input); // for any values we may have to pass
into it

  String get_name();
protected:
  PARAM_TYPE type;
  int id;
  String name;
};

// example probablistic button behavior
template <typename N>
class probs_behavior : public base_behavior<bool, N>
{
public:
  probs_behavior() = default;
  ~probs_behavior() = default;

  N process(const bool& input) {
    return probs[(index + 1) % 10];
  }

private:
  int index;
  N probs[10]; // if we wanted to have this be variable size, use
std::vector<N>
};
```

3.4.2.3. Wrapper Class and Application

We currently have two disjoint objects that are important to this system, but we would probably like to have their functionalities together in one spot so we do not need to call methods from multiple objects to achieve one goal. Thus, we can create a `wrapper` object that combines the two together:

```
template <typename IN, typename OUT>
class parameter_wrapper
```

```
{
public:
  parameter_wrapper() = default;
  parameter_wrapper(param<IN> p, base_behavior<IN, OUT>& b) :
    param(p), behavior(b) {}
  ~parameter_wrapper();

  // may need a little more than this, but I think this gets the idea across
  OUT process(const IN& input) {
    param.read();
    return behavior.process();
  }

  // other methods like getter and setters are hidden.

private:
  base_behavior<IN, OUT> behavior;
  param<IN> param;
};
```

In the Arduino IDE:

```
#include "behavior.h"
#include "parameter.h"
#include "parameter_wrapper.h"
// ... other libraries

button button(3, "button pin", "test button");
probs_behavior<int> probs;
parameter_wrapper button_wrapper;

void setup() {
  button_wrapper(button, probs);
}

void loop() {
  Serial.println(button_wrapper.process());
}
```

There could be other extensions to this system, such as including multiple behaviors, or including "modifiers" that could lightly modify values as they enter a behavior, or including "modulators" that change behaviors or modifiers over time, or allowing the wrapper objects create parameter and behavior objects themselves, but I believe that for the sake of demonstration, the system above is simple enough to understand the idea that I intend to convey for the "data processing" box. There is an additional GUI system that also has to be constructed for the user to interact with, but I haven't had any idea as to how that would look like.

3.5. The Printed Circuit Board & Physical Components

The PCB was inspired by Mason Mann's less rigid designs that he has used in his various selfmade audio electronics projects. As my first printed circuit board, I think

it turned out well, if "turned out well" translates to properly alinging traces, ensuring every part has a correct footprint, and that every component can properly interface with the Teensy. **The number of headaches that have been associated with the design of this circuit board do not outweigh the aesthetic chaos that comes with the board.** It is subsequently obvious from my personal experience developing this board that I strongly discourage any designs similar to the liberties that I took.

This anecdote centers around the Hall Effect joysticks I bought from Aliexpress and its inability to print sensor information. There is an interesting pendulum that swings back and forth in regards to the parts that this prototype needs. On one hand, parts can be readily available for cheap on websites such as Mouser, Adafruit.com, Digikey, or Aliexpress. However, in choosing locations such as Aliexpress to source parts from, it becomes rather difficult to source proper datasheets for certain products. There was an issue I encountered with a hall effect joystick that did not print data to Serial even aftewr being properly wired. After investigating the power trace on my PCB I noticed it was connected to the 5v pin isntead of the 3.3v pin most components are designed to take. After rewiring the power wire,the hall effect joystick in question still was not able to print values out to Serial.

Additionally, in the design like this, I found that more of my attention was taken away from ensuring that the PCB itself was properly put together (since I also accidentally flipped the display's footprint, leading to another headache resoldering wires to the proper pins. Had the PCB been a little bit more neat, there would most certainly be a clearer procedure to refer to when troubleshooting potential causes.

3.6. The Ambitious NYU Student

It's likely that because you are reading this part of the report you're just as interested in audio hardware, software, or audio embedded systems as I am. If I were to guess, you are also probably looking for anybody or anything that will satisfy this thirst of knowledge, an interest for the design patterns and theory through the lens of art. There's already a lot to take away from this giant word vomit, but it's also important to acknowledge the following:

1. It's likely that if you are interested in taking the Product Design class, you have no idea where to begin in terms of fabrication and enclosure design. **Do not make more work for yourself if this is the case.**
2. **It's very normal to make compromises in this class.** I had to make a lot of compromises throughout the semester because in addition to the Product Design class, I took Basic Algorithms and Operating Systems, the notorious weeder classes in the Courant Computer Science department.
3. **It's not fair to compare yourself to others in this class.** See the second point. If the first point also applies to you, there is no reason for comparisons or an unending amount of anxiety that you are falling behind. Lower the bar, nurture your ideas and the soul. They will all come eventually. Having gone through this semester, I don't think it is worth the added stress of getting everything done at

once compared to the amount of sleep and caffeine that you will lose (at least from my experience)

If it's not clear yet, be sure to **set your bar lower than you'd think**. Not for the sake of putting you down, rather for the sake of your mental health and sleep schedule, unless you really know what you're doing.

4. Fabrication

4.1. Design Goals for Future Prototypes:

My product will functionally look similar to handheld gaming consoles, namely the Nintendo Switch, the Wii-U, and the Steam Deck, shown below. Future iterations on this project involve on 3D printing an enclosure (smaller than my current breadboard layout)
to ensure that the device maintains a small formfactor and is handheld. I might need to include additional 3D printed components (for the DPAD, or a custom joystick footprint).

Figure 1: The Valve Steam Deck (left) and Nintendo Switch (right)

Figure 2: The Nintendo Wii-U

4.2. Inspirations

My product takes inspiration from the Y2K “translucent tech” design, and the transparent yet minimal asthetic from tech companies like Nothing and Teenage Engineering.

Figure 3: The Nothing Ear 2 (left) and Nothing Ear 3 (right) wireless earbuds

Figure 4: The Nothing 3a Pro phone lineup

Figure 5: The Teenage Engineering TX-6 and OP-1

Figure 6: Y2K translucent style tech products. the Apple iMac G3 (left), Nintendo GameBoy (middle), and Nintendo GameCube (right)

4.3. Rough Sketch

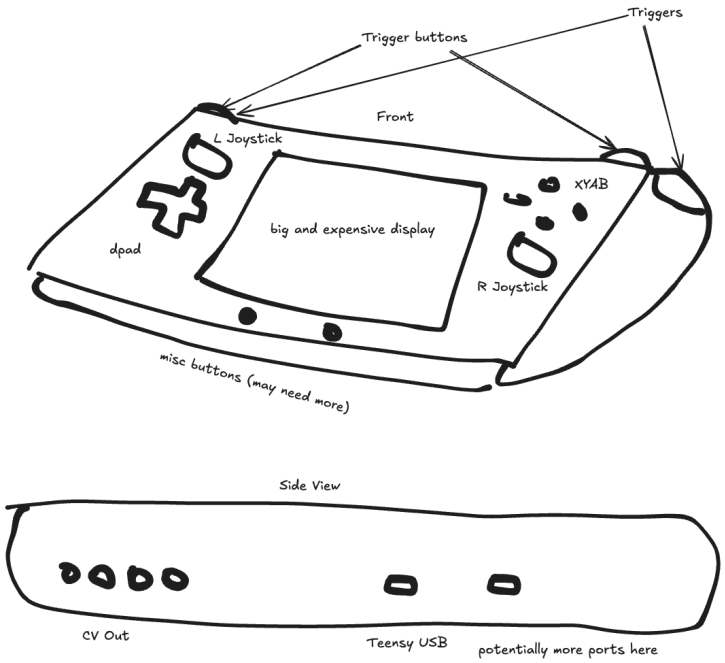


Figure 7: rough sketch of my game controller

5. Planned Features

There are a plethora of features that I plan on adding in the future. Below is an incomplete list:

- ☐ Better documentation of sourced parts
- ☐ Code Refactoring for better Object-Oriented Design
- ☐ Menu / GUI Infrastructure
- ☐ Asynchronous `analogRead()` using `enableInterrupts()` and `disableInterrupts()` from the Teensy ADC class.
- ☐ FrameBuffer display implementation
- ☐ Bluetooth Support
- ☐ Lithium-ion battery power supply option
- ☐ PlatformIO migration
- ☐ tinyUSB stack implementation for thorough USB-Compositing
- ☐ XInput Option via tinyUSB.

Modular-based Data Processing Firmware Design:

- ☐ Behaviors
  - Allows the user to set the ways in which a parameter behaves. One per parameter, per output must be allowed.
  - For instance, a button can be a simple toggle switch (on/off), trigger a random value upon button press, cycle through an array of values, probablistically trigger a set of values, etc.
  - Similarly, potentiometers can also simply display the X and Y values, a polar coordinate, or a logical computation between the two (print X if X>Y, Y otherwise), etc.
  - Far-fetched behaviors:
    - physics simulations (each bounce of a marble in a container is an output of some kind)
    - time-stretching (ie the incoming values for a parameter will be slower by a given factor)
    - logic gates (ie a button will only output a value v if conditions A and B are met. If we assume condition A is when the button is pressed and condition B when the gyroscope detects very little movement, we can easily create very dynamic conditionals for data to be sent.)
- ☐ Modifiers
  - Allows the user to fine-tune the parameter's pre-behavior / post-behavior values to their liking. Aggregates in a tree data structure.
  - for instance, a button can obtain the behavior to send values $a, b,$ or c as an output value. The user can set what each value is, and whether or not he wishes to bias an output (ie introduce a coefficient that will be summed or multiplied with the output), slew the output (if a was sent out first and b is the next value, we interpolate between the two discrete values), or invert the output.

- similarly, a potentiometer can obtain the behavior to send the polar coordinates of its respective x and y potentiometers. It can decide the smoothing value, bias the outputs using some mathematical function, or feed the outputs into a separate function of its own (ie if x and y are phase or frequency parameters for a Lissajous figure, and the output of the function is the phase or frequency relationships of the two)

☐ Modulators

- additional functions that control different modifiers / behaviors

(in the scenario of the button above, if we had a way to deterministically change which values a, b, c were being outputted in addition to the button trigger, or a way to modify a, b, c by some amount)

- audio thru modulator that lets the user control parameter behaviors

with audio (a button press gets frequency modulated with a incoming audio signal)

6. Materials List

6.1. Hardware

Component	Notes	Count	Price
Full size Breadboard	for prototyping, can be from any vendor.	2	$11.9
Analog 2-axis Thumb Joystick with Select Button	These can be replaced with hall effect sensors as mentioend above. The ones I got from Aliexpress were from looking up “hall effect joystick” and picking one that wasn’t expensive	2	$11.9
Soft tactile push buttons (20pk)	These are the ones I ended up using but any push button will do	1	$2.50
MCP4728 4-channel DAC	recommended by Steve	1	$7.50
LSM6DSOX Gyroscope	probably overkill but any gyroscope can work	1	$11.95
PJRC Teensy 4.1	Microcontroller of choice	1	$29.60
10k Breadboard Trim Pot	Act as triggers, can be buttons need be. You can also find these on Aliexpress by searching for Hall Effect Axis Resistor	2	$2.50

Component	Notes	Count	Price
2.2″ 18-bit color TFT display	Any 2.2″ TFT display should work, doesn't necessarily need to be from Adafruit.	1	$24.95

6.2. Software

Library	Notes
Adafruit ILI9341	Adafruit driver library for the ILI9341 display.
Adafruit MCP4728	Adafruit driver library for the MCP4728 DAC.
Adafruit LSM6DSOX	Adafruit driver library for the LSM6DSOX.
`Keyboard.h`	PJRC Keyboard library
`usb_midi.h`	USB MIDI PJRC library
`Wire.h`	Dependency for adafruit libraries
`SPI.h`	Dependency for adafruit libraries
Adafruit GFX Library	GFX library used during most of prototyping.
ILI0341_T4	**Requires manual installation**. Framebuffer optimized for Teensy 4.1 and ILI9341 TFT displays. **Use this if you intend on using the tgx graphics library.** Never fully implemented. **Remove the Adafruit ILI9341 driver if you plan on using this one.**
tgx, a tiny 2D/3D graphics library	**Requires manual installation**. Graphics library optimized for ILI9341 TFT displays. Can display simple shapes, curves, images, animations, and 3D shapes. Never fully implemented.
ResponsiveAnalogRead	**OPTIONAL**, if you do not want to implement asynchronous analog reads, or if you do not care about this micro-optimization.
Adafruit GFX Buffer	**OPTIONAL; requires the Adafruit-ILI9341 driver**, if you want a framebuffer for the Adafruit GFX library.
Arduino XInput	XInput library for microcontrollers like the PJRC Teensy 4.1, for later implementation. Emulates Xbox 360 communication protocol. Requires the installation of the following