Engineering Capstone Project Part B

Final Report
‘Developing a Synthesiser Control Board’

Students
Chris Deakin s3296014
Dan Ross s3688083

Supervisor
Samuel Ippolito

Executive Summary

This project involved the design and creation of a synthesiser control board named the Heron, built to interface with portable synthesisers to create music. The goal was to have a fully functioning control board that could be used live, in the studio, and must be able to run on battery power so it can be played anywhere.

Initially, criteria were defined, and sketches were done on both the physical board and the UI elements of the screen. The criteria were brainstormed by the team after research on what other products in the market have, what parts felt useful or not, and new features the team thought would complement their own playing, while being lightweight, portable and being able to run without additional external feedback such as a computer. Prototyping was done on breadboards to test aspects of the features before committing to the design.

On completion of the design, work was split between the building and coding of the physical parts of the board and the design and coding of the screen UI to complement the physical aspects of the board. Physical features and components were built early on as those features were set and the code was built after and around them. The screen was built in isolation to the board but was tested frequently to ensure smooth running between the two devices.

Along the way small modifications were done to the device by changing the UI slightly, introducing ad-hoc connections to fix circuit board issues and leaving some features unfinished due to time constraints on fixing them.

On final integration of the screen and board, bugs were found generally to do with the screen UI having graphical glitches and the LEDs not displaying correctly. While the UI was fine tuned to a working state showing all relevant material, the LEDs had some hardware issues that, due to time constraints, could not be fixed.

In its final state, the Heron successfully integrates with the Axoloti synthesiser to control the effects generating and modifying audio. It runs completely independent from a computer, save loading patches to the synthesiser, and meets all the criteria set out at the start including non-exhaustively working encoders, analog devices, sequencing, saving and loading, displaying relevant information and functioning as a simple keyboard.

As such, the performance of the Heron is considered successful, and functions as was set out from the beginning.

Group Member Contribution Table

Section	**Person Responsible**
Executive Summary	Dan Ross
Introduction	Chris Deakin
Background and Literature Review	Dan Ross
Methodology	Chris Deakin
Design	Chris Deakin / Dan Ross
Results, Conclusions, and Future Works	Chris Deakin

Introduction

The Heron is a ‘synthesiser control board’ designed to interface with the Axoloti synthesiser. The Axoloti is a powerful digital synthesiser, but lacks the user interface to make it a useful standalone product for musical production and performance. This limits users of the device either to those with the technical knowledge to provide their own hardware, often rudimentary, to use the Axoloti.

The Axoloti patcher software can be used to create a practically infinite number of audio processing algorithms, or ‘patches’, through a visual interface that requires little technical knowledge. The Heron aims to accompany this with a user interface, in hardware and software, that makes live control of the created patch accessible to a wide audience.

The Heron has a touch screen, eight encoders, two sliders, a joystick, and 32 keys. The encoders, sliders, and joystick are primarily used to control variables within an Axoloti patch; the keys can be used either as a keyboard, to directly play notes, or as a sequencer, to arrange patterns of notes that play at a user-adjustable speed. The touch screen is primarily used to display information to the user, and allow for the control of certain on-board values (such as assigning encoders to a particular element, or saving and loading user presets).

The Heron is a 340x150mm board built around an ESP-32 microcontroller. The code is written in C/C++, and implemented using the FreeRTOS operating system. The code has been split into various tasks to handle different aspects of the system, and interface with peripherals like I2C, SPI, I2S, & ADC.

The project was split into a number of distinct parts, or features. The majority of these tasks were completed, while others were not attempted due to a lack of time. The core features for interfacing with the Axoloti were prioritised, and the features for interfacing with other external devices were abandoned.

The Heron ultimately delivered a user-friendly device for interfacing with the Axoloti. While a number of planned features were not developed, these were not core to the function of the device, and the structure of the code (particularly as it relates to the operating system) allows for easy implementation of new features in the future.

Background and Literature Review

Electronic music has spanned over three centuries since its first recorded appearance in the nineteenth century to, perhaps more widespread, the mixing and mastering of almost all songs in modern music. Holmes[1] states that electronic amplification of voice, predating the telephone, was capable of amplifying tones within an octave, perhaps by some definitions being the first electronic instrument. A more modern accepted definition of an electronic instrument may be Elisha Gray's "Musical Telegraph", a machine that had reeds and keys such that on pressing the key, an electrical circuit closed which would make the reed vibrate at a certain frequency that emulated musical notes.

As history moved to a more modern age, advancements included true tone synthesis from the Teleharmonium[2], the first tape recorder[3] and thus the first speed, pitch and mix manipulation using tape, the analog synthesisers such as those created by Bode and Moog, and to where the world is currently using high speed computers to digitally manipulate samples, generate waveforms, perform intense effect changes all while having the capability of running live and with relatively low power usage.

The artistic movement behind the innovations has been just as historied as the devices themselves, with the Teleharmonium inventor Thaddeus Cahill seeing his invention the perfect way to harness the commercialisation of his invention and music reaching ears right around New York City. With headlines reading "Get Music on Tap Like Gas or Water", his model was to get people to subscribe to his sessions, getting lucrative deals with hotels around Broadway[2]. Across the globe the Futurist movement in Italy predicted the evolution of electronic music whilst using it and the microtonal capabilities to step away from the heavily storied past of Italian music. Post-war avant-garde music opened way to electro-acoustic movements such as those created by John Cage, whose influence and works have been referenced and sampled by artists as early as Brian Eno[5] in the 1970s, to Aphex Twin in his later works, to name just a few. The more poetic forms of music gave way to the more popular, accessible subgenres and their evolution, from disco's popularity in the 70's with its for-on-the-floor beat, to incorporating keyboards heavily in rock music through the 60s to the 80s, further still to trance and the more indie Euro subgenres of pop until our current time which, as mentioned before, involves some kind of electronic influence whether the instruments or mixing in every song.

While synthesisers such as those Moog put out initially had immense customisability for the time period, they were quite large, had a large number of wires and knobs to fine tune, and were expensive. Following in this modular vein, modern day synths such as those in the Eurorack format feature the same kind of technology but at a lower price and smaller in size. These synthesisers are analog, designed to modulate and route waveforms to different parts of the synthesiser. You can take a sine wave, send it simultaneously to a delay and a filter, and use the filtered wave to control the delay time, for example, whilst also turning knobs and rerouting cables. These modules allow for an almost unlimited amount of adjustment to the sound they can produce, given the space. The Arturia Keystep[10] series is an example of a modern keyboard controller that is common with this technology, but it is common to simply play with just sequencing and drum machines.

Modern computers can not only emulate these synths quite well, although in a digital space, they can allow the user a much faster recording space, with access to multiple tracks at the click of a button, fully automated effects and tables of waveforms to run through. Samples can also be loaded, with some plugins able to host terabytes of information as recordings and play them as though playing live, with each note transition, each velocity, each aspiration or

modulation and movement all customisable, allowing for creating multi track symphonies for anyone with access to a computer.

Devices like the Novation Launchkey[6] or the Akai MPK[7] series are pure controllers, controlling only the software on the computer that accepts MIDI signals. These offer a number of encoders, sliders and sample pads, with basic menu interaction and sequencing available. Other products like the Novation Launchpad[8] are designed with live playing in mind, playable with one hand to play samples or notes in a scale, highly customisable and integrates well with software. The Ableton Push 2[9] completely integrates with Ableton Live, having a fully featured menu, complete customisability, and the ability to play without even needing to look at the computer, but still needing to be plugged into its host.

The technology exists to interface with these two different areas of electronic music, but the niche of having something small, lightweight and still very customisable is an area not fully explored. The Eurorack standard simply can't be used in a portable environment because of its size, and it is a difficult area to get into as a hobby due to its cost. The digital side of using computers does offer portability but all the peripherals require connection to the host which is an expensive laptop that has to be open and on to run.

Access to these kinds of technology in the modern era has allowed anyone to access electronic music regardless of age and with services like SoundCloud, are able to get that music out in the world for anyone to listen to, something Thaddeus Cahill could truly be envious of.

Methodology

The general process used to develop the Heron, both as a whole and for individual aspects, was as follows:

- General design work undertaken as a group.
    - The project or feature is discussed by both group members. The desired functionality is discussed, and a general design is agreed upon.
- Tasks are identified and distributed.
    - The necessary work is, as far as possible, broken down into discrete tasks that can be assigned to a group member.
- A timeline is created for the tasks.
    - An agreement is made as to when a task or tasks should be completed, based upon the importance of the task, the expected time required, and what other tasks depend upon its completion.
- Meeting/s are had to update on progress.
    - Group members discuss the progress on the individual tasks, share completed work (if possible), and update the timeline on tasks if necessary.
- Individual work is integrated.
    - Work done individually on different areas is integrated into the project proper, if ready to do so.
- The process is repeated, as needed.

For smaller, more incremental work done throughout the year, this process was done mostly informally, through communication between the group members. At several points, however, larger discussions were had in this light as to the progress of the project, and several timelines were produced.

While technical solutions were generally developed individually, the design of the product, or individual features, was undertaken as a group. For the Heron as a whole, this happened in two ways. At the beginning of the project, meetings were held to consider the desired features, particularly as they relate to the expected use cases, and users, of the device. Key features were identified, with hardware and software choices made around them. Optional features were discussed and planned on the basis of how much they would add to the product in comparison to the work required.

For individual features, less formal meetings were required, but discussions were nonetheless held as to what the expected function of a feature was, what aspects should be prioritised to make it functional, and what further parts could be included if time allowed.

An initial timeline (Appendix C) was produced at the beginning of the project, with weekly intervals estimating the needed time and expected completion date of various tasks, on weekly intervals throughout the year. Not all tasks were distributed at this point, as it would be exceptionally unlikely that any such distribution would hold for the entire year. Instead, regular meetings were held, in which work was distributed based on current circumstances.

The initial timeline was not expected to remain accurate or relevant for the entire year, however, and further timelines were developed. At the end of the second semester, a new timeline was developed (Appendix D). Completed tasks were removed, and some tasks that

would no longer be attempted were dropped. This second timeline was more focused than the first, and contained more features than low-level tasks.

In the second semester, a final timeline was made (Appendix E). At this point the group had more experience on the project, and working as a group. Additionally, the number of remaining tasks had dwindled, and only two months remained on the project. For these reasons the timeline proved to be more accurate than the previous two, and tasks were assigned immediately.

Fortnightly meetings were held at a scheduled time with the group supervisor, and more frequent meetings were held irregularly with the group members. It was not possible to have these meetings in person, so online alternatives were used. Supervisor meetings were generally used to update on the general progress of the project, and to ask for advice as a group. More technical details of individual tasks were generally brought up in the meetings between group members. Tasks were mostly approached individually, but assistance would be sought frequently on specific problems, as required. This allowed work to be apportioned effectively, without individual members becoming stuck on an issue.

A private Github repository was created to synchronize and share the code written individually. A primary project was created to contain the combined work for the project, but other projects were created as needed to test and work on some specific areas. As an assembled board was not available for both group members, some work was required to integrate the work done off-board (primarily on a bread-board setup) into work deployed to the actual Heron hardware. The ability provided by Github to easily share code from these two environments, merge that code into a main project, and keep track of changes made by group members, was extremely useful in development.

Design

Initial Design

At the beginning of the project, a general understanding existed as to the problem to be solved, and the sort of product that would be produced to solve that problem. As discussed previously, the Axoloti device existed as a powerful, cheap, portable product, but was not viable for most users due to its lack of user-interface (being limited to a software 'patcher' that is sufficient for developing patches, but unappealing for playing them).

The Heron was thus envisioned to be a portable device that provided user interface to the Axoloti such that a casual user could easily produce music with the Axoloti without needing to develop their own hardware. It would have dedicated hardware interfaces, such as encoders, in sufficient numbers to allow for controlling a number of variables simultaneously, but with the understanding that it is aimed at users who are unlikely to be controlling dozens of elements at a time. It would allow for a degree of flexibility in its operation such that it did not overly limit the near endless modularity of the Axoloti. It would also have a number of external connections for interfacing with devices other than the Axoloti, and to act as a bridge between the two.

As initially conceived, the Heron would have eight encoders, two sliders, a joystick, 32 mechanical keys, a screen (potentially a touchscreen), and a number of generic buttons. The sliders and joystick, being analog devices, would require developing a second method to obtain data that would be used in the same manner as the encoders (which were read via I2C). This was considered a reasonable cost for providing the user with additional methods of control that may be useful, or enjoyable. The sliders providing a physical maximum and minimum, and the joystick allowing users to adjust two values (one for each axis) relative to each other in a manner that is theoretically possible with an encoder, but not as practical.

The keys would be laid out in two rows of 16 keys, for use both as a keyboard and as a sequencer. The use of mechanical keys allowed for this mixed functionality, as opposed to keyboard keys, and would feel better to use than a more generic button. The buttons that were included did not have fixed uses in mind (other than the octave buttons on either side of the keyboard), but were included to allow for various user interactions not yet decided upon.

The screen would be used to convey much of the information to the user, such as the current value on an encoder or slider. A touch screen was considered to be ideal, as navigating the various possible states with just the buttons could prove to be annoying at best or confusing at worst.

The external connections planned were: a pedal input, which would likely take an expression pedal, but could be used for other analog inputs if desired, to act as the sliders or joystick would; miscellaneous outputs, connected to an IO expander, to allow for low intensity digital outputs such as a 'gate' signal to another synthesiser; MIDI input, likely for the connection of a keyboard if the user did not wish to play on the mechanical keys; and a MIDI output, which would allow the Heron to also control other devices, if they were compatible.

A large number of LEDs were also included, to quickly display information to the user, such as if a button had been pressed, or to give a rough indication of the value on an encoder.

Hardware Design

Under normal conditions, it may have been preferable to finalise and order the PCB for the Heron further into the project. Given the global situation during development, however, it was decided to prioritise getting a prototype ordered as soon as possible; the speed of delivery and availability of parts in the future could not be guaranteed, and so hardware was ordered halfway through the first semester.

A four-layer PCB, 34x15cm in size, was designed. The layers were, from top to bottom:

- Signal layer. The majority of components were placed on this layer, as the manufacturer only offered assembly on a single side. Primarily horizontal traces were run on this layer.
- 5V layer. This was a mostly continuous copper pour across the board. While a 3.3V layer could also have been useful, the large number of 5V LEDs across the entire board made a 5V layer a better choice.
- Ground layer.
- Signal layer. The key hotswaps (discussed below), jacks, and vertical traces were on this layer. A final version of the PCB would have the Axoloti and power headers, as well as the ESP on the bottom side of the board. For the prototype, however, it was more convenient to have these on the top.

Rather than having the microcontroller directly on-board, which would require a considerable amount of extra work for minimal gain, a development board was used, connected to a series of headers on the board.

As can be seen in figure 1, through holes were placed on either side of the development board for easy access to the pins. It was expected, and proved correct, that the traces on the board would not be final in all cases.

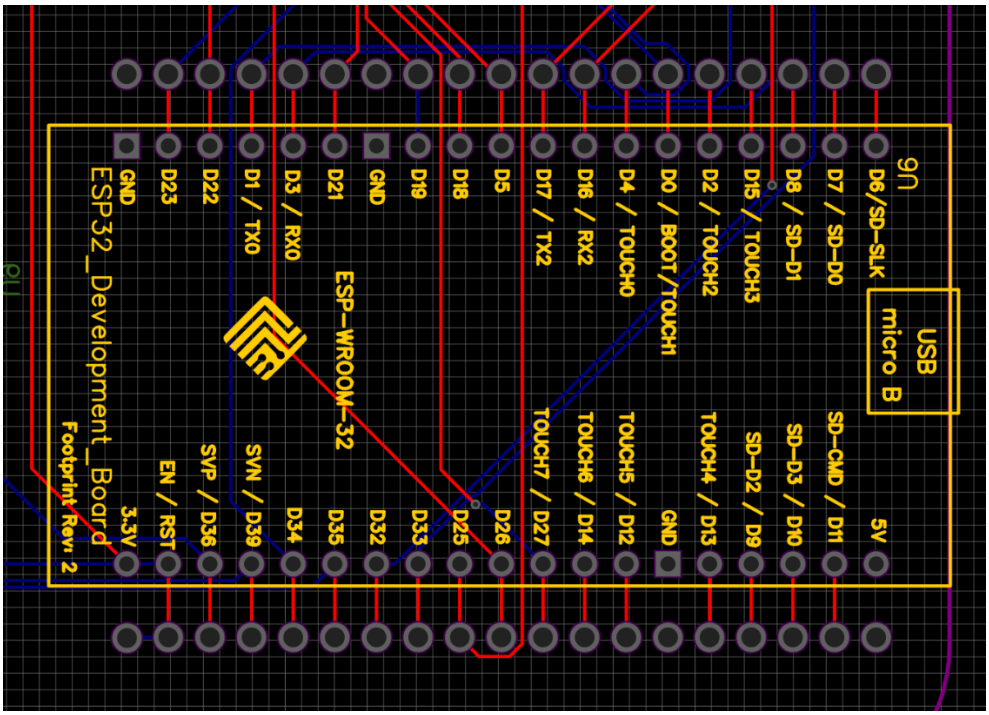


Figure 1, Devboard layout on PCB

As the Heron would have eight encoders, each requiring two IO pins, and a 7x8 button matrix, it was not practical to directly use GPIO pins on the microcontroller for these parts.

Instead, MCP23017 IO Expanders were used[11], connected via I2C (an SPI version of this IC is also available). The IO expanders have 16 pins, organised into two ports, have three address pins (starting at a base address of 0x20), allowing for up to eight such devices on a single I2C line. They also have an interrupt pin for each IO port. It is possible to mirror the interrupt pins so that one pin will serve both ports (in order to economise on the number of pins used). The interrupt pins did not prove needed, however.

To bridge the 3.3V data of the ESP32 to the 5V IO expanders and LEDs, LSF0102 Logic Level Converters were used[12]. These ICs allow for bi-directional, two channel conversion between two reference voltages (in this case, 3.3V & 5V). As can be seen in figure 2, taken from the datasheet, these logic level converters do degrade the signal. This would not prove to be an issue with the I2C line, but did seriously interfere with the operation of the LEDs. Additionally, the part chosen became impossible to source commercially as the boards were ordered. It was through chance that second-hand ICs were found from an alternative source that were used on the boards - similar ICs were available, but in different packages to the one used on the Heron, so using an alternative part was possible but using them would be difficult and precarious.

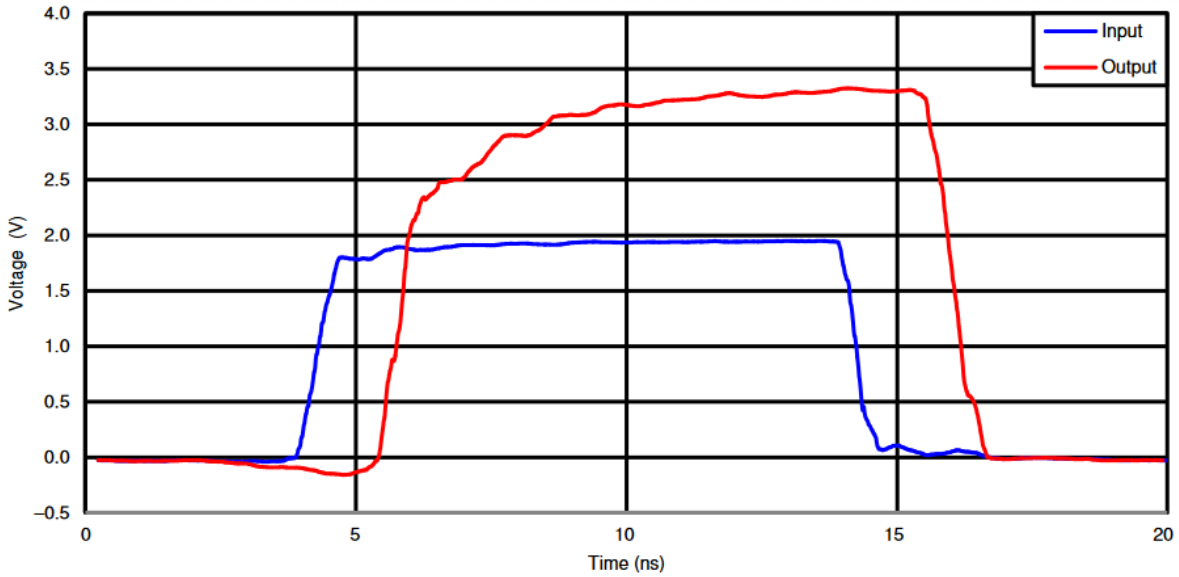


Figure 2, LSF0102 voltage translation characteristics (Fig. 6-1, [12])

Debouncing for the encoders was performed in hardware, with the use of schmitt triggers[13], shown in figure 3. Debouncing the button matrix was performed in software, however. The nature of the quadrature encoders is such that it is important to keep an accurate track of their previous readings. Ascertaining the direction of the encoder turn was performed by comparing the current state of the pins to the previous states, thus erroneous pin readings could easily be an issue. For this reason, it was felt to be important that hardware debouncing was included for the encoders. The mechanical keys, however, did not have this need (and were found to have bounce periods of only a few ms, at most), and as such software was thought sufficient. In hindsight, however, some minor key bounce was occasionally noticeable during development, so with the miniscule cost of components required for hardware debouncing, it would have been advisable to do so for the keys too.

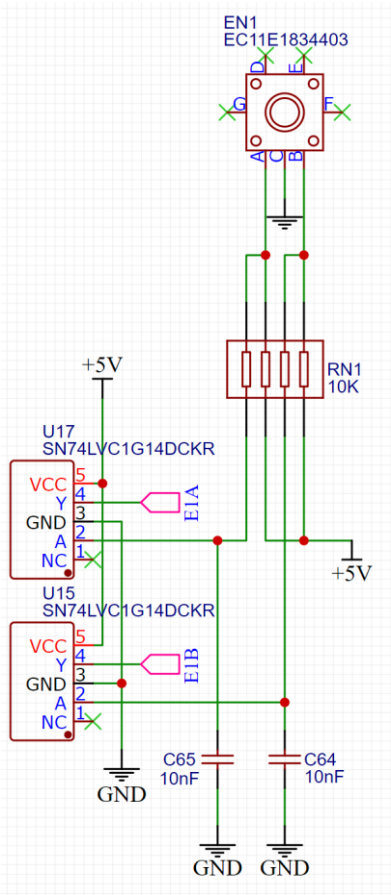


Figure 3, Encoder Debouncing Circuit

The manufacturer's assembly service handled most of the surface mount components on the board, aside from the logic level converters. The encoders, joystick, keys, and buttons were all attached after delivery, as were female headers for the screen and ESP-32, and male headers for power and the connection to the Axoloti. 'Hot-Swap' pads for the keys were soldered on the rear of the board. These pads (figure 4) sit on the reverse side of the keyboard, and allow keys to be easily inserted and removed.

Figure 4, Mechanical Switch 'Hotswap' Pad

Two manufacturing faults were detected, however. An IO expander had three pins severed, as seen in figure 5. Replacing this IC, and attaching the logic level converters, required the use of a preheater pad, solder paste, and hot air gun - a more complex operation than the through-hole components, which could be done with a soldering iron. Additionally, one LED appeared to have a poor connection, only receiving data when pushed down on with a finger. This LED was replaced entirely, as the process of redoing the connection subjected it to high temperatures for long enough to potentially damage it. As spare LEDs were available, it was felt safer to simply replace the unit.

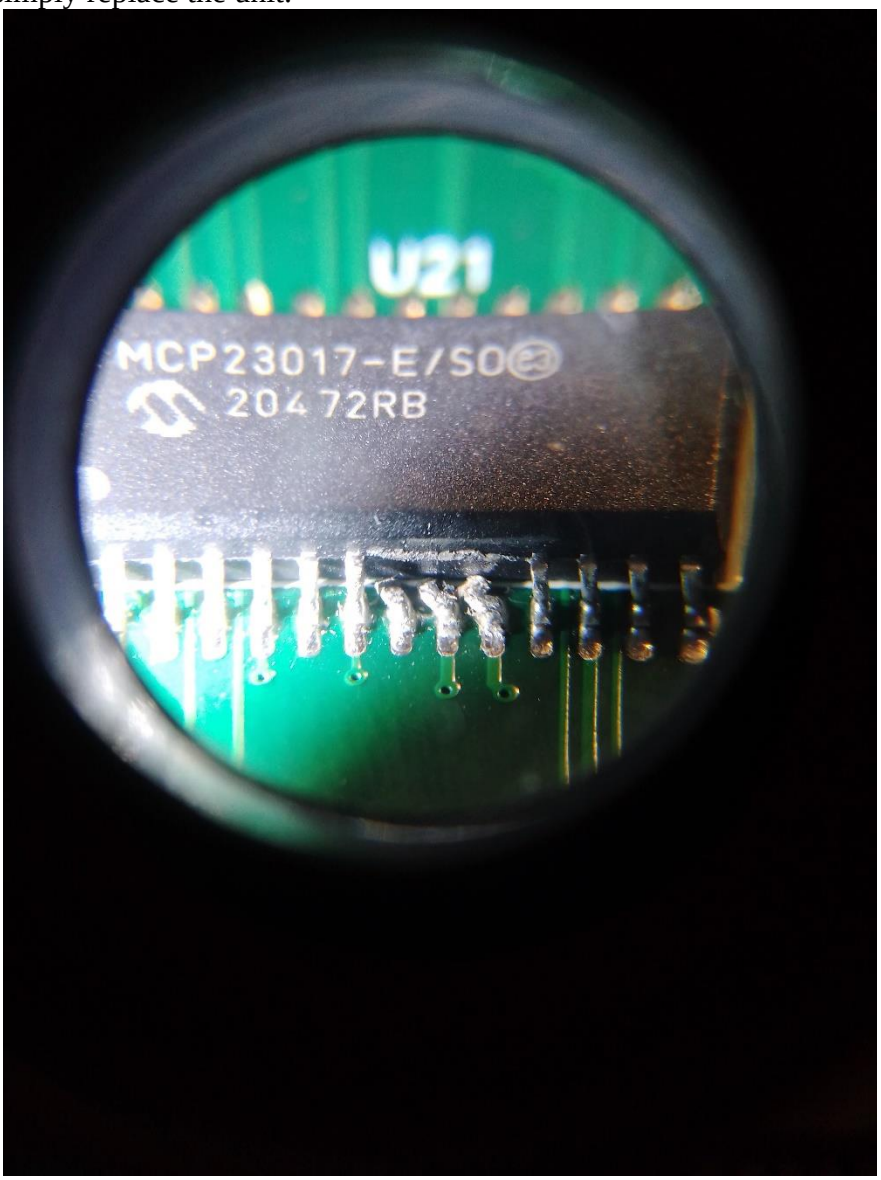


Figure 5, Damaged IO Expander, as delivered

Initial Programming Work

Programming work was started prior to the delivery of the PCBs. ESP-32 development boards are widely available, and several were purchased or were already on hand, along with encoders, buttons, screens, and joysticks. This meant that work could begin immediately on some basic components. Development was conducted within the ESP-IDF framework, produced by the manufacturer, Espressif. An automotive focused environment also exists, ESP-ADF, as is the Arduino framework for ESP. While the Arduino framework would save some time with basic components, this would come at a cost of efficiency to the overall system, and may well have resulted in more difficulties in later development.

The earliest programming work done for the project was on setting up the communications protocols used in the Heron: UART, I2C, & SPI. The UART component was the simplest of the three communications protocols, and would not be needed until more specific MIDI code was written. The process for configuring UART largely follows the process for configuring other protocols in ESP-IDF (see figure 6). A config structure is created and attached to a particular port, the pins for the port are set, then the driver is installed with the previously configured information. UART communications, in their simplest form, require only the 'uart_write_bytes(port, buffer, bytes to send)' function.

```
const uart_port_t uart_num0 = UART_NUM_0;

// Create a config type with all the settings we want
uart_config_t uart0 = {
    .baud_rate = UART_0_BAUD, // Defined this in MIDIdefs.h
    .data_bits = UART_DATA_8_BITS, // The rest are defaults
    .parity = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_DISABLE, // Except disabled flowcontrol (only using TX/RX pins)
    .rx_flow_ctrl_thresh = 122,
};

uart_param_config(uart_num0, &uart0); // Associate the config we just defined above with UART0

uart_set_pin(UART_NUM_0, UART_0_TX, UART_0_RX, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE); // Set the TX/RX pins for UART

QueueHandle_t uart0_queue; // Presumably handles UART queue

uart_driver_install(UART_NUM_0, UART_0_BUF_SIZE, UART_0_BUF_SIZE, 10, &uart0_queue, 0); // Finally install the driver
```

Figure 6, UART Driver Installation

The I2C drivers are set up in much the same way as UART, but the process for sending and receiving messages is more complex. Figure 7 shows a single message sent via I2C. A 'command link' must be created, taking control of the I2C line; the 'write_byte" functions are then used not actually to send the data, but to assemble a message to be sent. The 'i2c_master_cmd_begin()' function actually begins the transmission.

```
i2c_cmd_handle_t cmd = i2c_cmd_link_create(); //Create a command link for the fo
i2c_master_start(cmd); //Start
i2c_master_write_byte(cmd, (this->address << 1) | I2C_MASTER_WRITE, ACK_EN); //S
i2c_master_write_byte(cmd, reg, ACK_EN); //Send the register address we want to
i2c_master_stop(cmd); //End
esp_err_t ret = i2c_master_cmd_begin(I2C_M_PORT, cmd, 1000/portTICK_PERIOD_MS);

i2c_cmd_link_delete(cmd); //No longer using command link
```

Figure 7, I2C Message Construction

As the I2C devices were accessed through an IO expander, further work was required in structuring the communications. The general process for communicating with the MCP23017 is:

- Send device address byte, OR'd with either the write or read bit. This selects the particular IO expander to communicate with, and tells it whether the microcontroller will be writing to, or reading from a register.
- Send register address byte. This tells the IO expander which of its registers the following operation will access.
- Send write data, or read incoming data. The IO expander will either write the new data to the given register, or send the contents of that register back out, depending on whether a read or write operation was indicated in the initial message.

On startup, after installing the I2C drivers, the ESP sends a series of messages to the IO expanders to configure them. These are:

- Setting the configuration register such that addresses do not increment after a read or write operation.
- Setting the internal pullups of each pin.
- Setting pins as input or output.

The value of the last two settings differ based on the device (for example, the button matrix will use one axis as inputs and the other as outputs).

No serious difficulties were encountered at this stage of the process. The communications protocols are all well documented, and could be tested without the final hardware on breadboard setups (see figure 8, where an IO expander was tested with several buttons and an LED). With these basic building blocks, and some familiarity gained with the ESP-IDF framework, further development could take place.

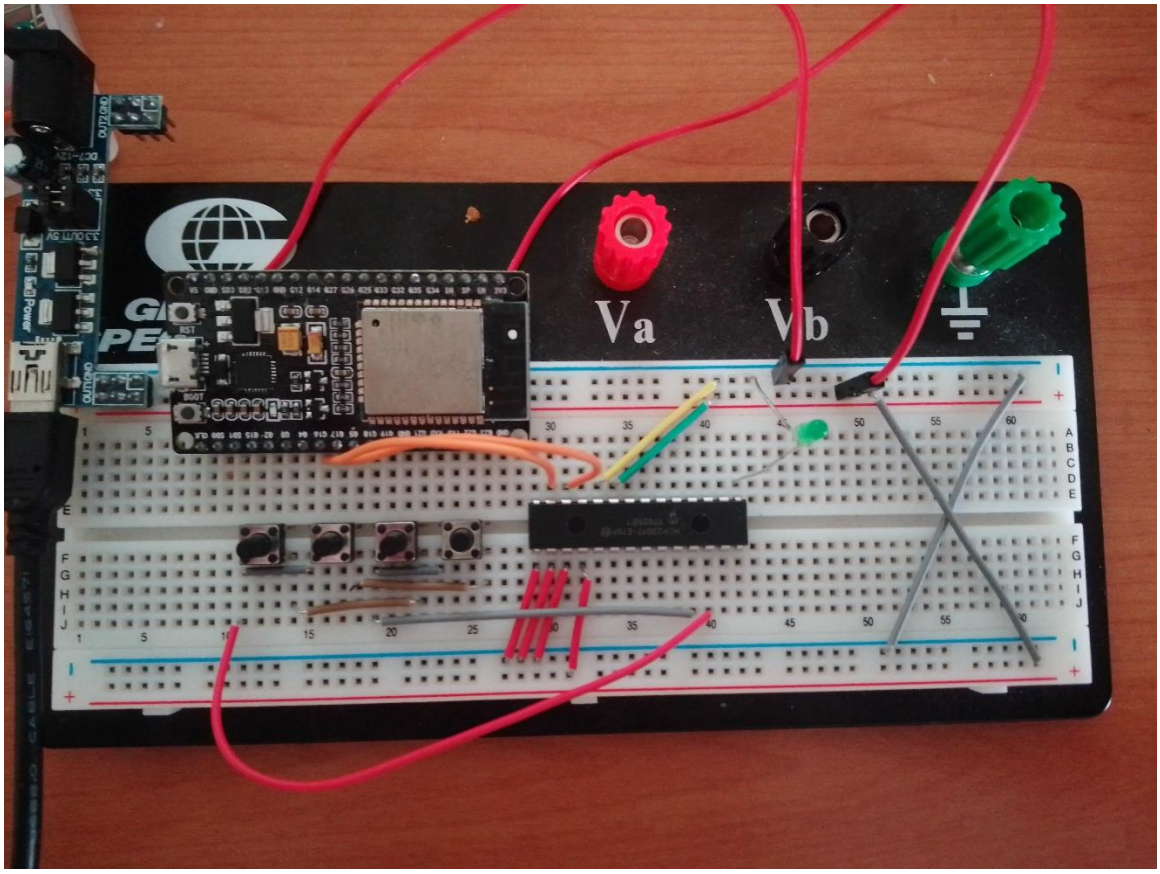


Figure 8, IO Expander Breadboard Testing

Feature Development

IO expander classes

The IO expanders, and their connected components, were managed through a series of classes. A parent class (figure 9) contains the elements common to every expander: a device address, configuration values, storage for the last read value on each IO port, as well as functions to setup the IC, and read from or write to registers.

```
class MCP
{
    protected:
        uint8_t address;    // Device address
        uint8_t config;     // IOCON value
        uint8_t PAdir;      // Port A input/output directions
        uint8_t PBdir;      // Port B input/output directions
        uint8_t PApu;       // Port A pullups
        uint8_t PBpu;       // Port B pullups
        uint8_t PAval;      // Port A current value
        uint8_t PBval;      // Port B current value

    public:
        MCP(uint8_t addr, uint8_t conf, uint8_t PAd, uint8_t PBd, uint8_t PAp, uint8_t PBp); // Constructor -
        void setup();                               // Will configure this IO expander with config, direction,
        void regWrite(uint8_t reg, uint8_t val);    // Writes a value to a register on the expander
        void regRead(uint8_t reg);                  // Reads a value from a register. If PA/PB will store in a

};
```

Figure 9, IO Expander Parent Class

Child classes were created for the specific expanders (figure 10), which contained operations such as reading the button matrix, or variables used in determining the turn direction of the encoders.

```
// Child class specifically for button matrix IO expander
class MCPB : public MCP // Inherits from MCP class
{
    protected:
    bool matrixState[56]; // Has an array to store the current state of all keys in the matrix

    public:
    using MCP::MCP;       // Need this to inherit the constructor from parent class
    void matrixRead();    // Function to read all values on the button matrix

};

// Child class for encoder button expander
class MCPE : public MCP // Inherits from MCP class
{
    protected:
    bool EV[16];    // Each encoder has two pins that we read, 8 encoders total
    bool EVP[16];   // We also need to store the previous values for reading encoders
    uint8_t Turn[8] // Turn direction (if any) of encoder on last read

    public:
    using MCP::MCP;       // Need this to inherit the constructor from parent class
    void encoderRead();

};
```

Figure 10, IO Expander Child Classes

MIDI System

The MIDI code was quite simple. A task was created in the operating system (discussed later) which received MIDI messages from a message queue. It split the incoming data into a three-byte structure used by MIDI messages, and passed them to the MIDI_send() function. The task suspends itself if no messages are waiting, and is woken by other tasks when they pass on a message. The MIDI send function simply sorts the messages based on their type before

sending them - most MIDI messages contain three bytes, but two byte messages also exist. In order for Axoloti to receive MIDI messages via serial, the 'serial/config' (figure 11) object must be placed in the patch, and the baudrate selected. The MIDI standard specifies that "The hardware MIDI interface operates at 31.25 (+/- 1%) Kbaud…"[14], but there is no need to adhere to this limit when specifically targeting the Axoloti. Were the Heron expanded to control other devices, however, it may be necessary to adhere to the standard in this regard.

```
void MIDI_Task(void *pvParameter)
{
 uint32_t Q3buff = 0; // Buffer for queue 3
 uint8_t status = 0;
 uint8_t d1 = 0;
 uint8_t d2 = 0;

 for (;;)
 {
     while(uxQueueMessagesWaiting(Q3) != 0)
     {
        xQueueReceive(Q3,(void *) &Q3buff,10);

        status = Q3buff & 0xFF;
        d1 = (Q3buff & 0xFF00) >> 8;
        d2 = (Q3buff & 0xFF0000) >> 16;
        MIDI_send(status, channel, d1, d2);
     }
     vTaskSuspend(NULL);
 }
}
```

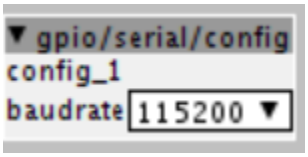


Figure 11, Serial/Config

Figure 12, MIDI Task

```
void MIDI_send(uint8_t type, uint8_t channel, uint8_t val1, uint8_t val2)
{
    char b1 = type | channel;
    char data[3] = {b1, val1, val2};


    if((type == CHNLPRESSURE) | (type == PRGCHANGE))
    {
        uart_write_bytes(UART_NUM_0, data, 2);
    }
    else
    {
        uart_write_bytes(UART_NUM_0, data, 3);
    }
}
```

Figure 13, MIDI_send function

No issues were encountered at this stage, figure 14 shows a hardcoded pair of messages being printed by the Axoloti console. The process of transmitting the messages themselves was straightforward, but difficulties would be encountered in other ways. MIDI messages do not convey audio data per se, but rather instructions for controlling the audio device. For example, the 'NoteOn' messages shown in figure 14 instruct the Axoloti that channel one should turn on note 100 (E7), with a velocity of 100. Once a note is turned on, it will remain on until a corresponding 'NoteOff' message is sent.

This would cause some issues in later development. Initially, the MIDI_send function was structured to send a note on, then a note off after a configurable delay, all within the same function. When only one system was using the MIDI task, this was not an issue, and as the various systems were generally tested individually, it was not noticed until near the end of the project that this solution was insufficient.

```
NoteOn  chl nl00 vl00
NoteOff  chl nl00 vl00
NoteOn  chl nl00 vl00
NoteOff  chl nl00 vl00
NoteOn  chl nl00 vl00
NoteOff  chl nl00 vl00
```

Figure 14, Axoloti MIDI Debug Messages

The issue arose when both the sequencer and keyboard tasks (discussed later) were used at the same time. Doing so would interfere with the delay between note on and off messages, meaning that notes would often become 'stuck' on, as their note off message had been lost.

To overcome this, a simple 'notes' class was created, linking a MIDI note to a state variable (figure 15). The state of a note tracks whether that note is inactive, whether a note on message should be sent, whether a note on message has been sent already (used when a key is being held down), and if a note off message should be sent. Rather than a simple delay, the variable control task checks the current state of all notes and sends MIDI messages accordingly. This operation is optimised in certain ways - for example, a note cannot be held down on the keyboard if it is outside of the currently selected octave, so notes outside that range do not need to perform that part of the check. An array of these objects, one for each note in range (0-127) was created.

```
class Notes
{
    public:
    uint8_t note;
    uint8_t state = 0;
    // State 0 - Nil
    //       1 - Send On
    //       2 - Sent on
    //       3 - Send off
};
```

Figure 15, MIDI 'Notes' Class

LEDs

The LEDs were perhaps the least successful element included in the final product. The LEDs could not reliably pass on data beyond (approximately) the 25th LED, out of a chain of 73. Beyond that point, the behaviour of the LEDs became unpredictable, and the LEDs were not entirely reliable before that point either.

The LEDs used were the WS2812C**[datasheet]**. These LEDs are intended to be placed in a chain, wherein the data for all LEDs is passed to the first, which takes its information (3 bytes, R-G-B) and passes the remaining bytes onto the next LED. These LEDs do not pass the unused data straight through, but have a reshaping circuit to preserve signal integrity. The capabilities of the reshaping circuit seem to be limited to minor distortions, based on the experience of this project. The WS2812C is a low-power (5mA operating current) version of the previous WS2812B LED, often sold as the 'Neopixel'. The datasheet of the 2812C claims full compatibility with the 2812B (see figures 16 & 17**)**. As can be seen, however, this was not entirely the case, with the centre time for a 0H or a T1L on the 2812B being higher than the top of the range for the 2812C. Notably, the minimum reset time for the B, 50µs, was considerably below that for the C, 280µs.

Data Transfer Time

T0H	0 code, high voltage time	220ns~380ns
T1H	1 code, high voltage time	580ns~1.6µs
T0L	0 code, low voltage time	580ns~1.6µs
T1L	1 code, low voltage time	220ns~420ns
RES	Frame unit, low voltage time	>280µs

Figure 16, WS2812C Timings

Data transfer time(TH+TL=1.25µs±600ns)

T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.8us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.45us	±150ns
RES	low voltage time	Above 50µs	

Figure 17, WS2182B Timings

The ESP-IDF port of the FastLED library was used to control the LEDs. This library had classes available for different models in the 2812 family, but did not actually make a distinction between them in timing. These timings proved generally compatible with the C's on the board, but adjusting them to be more in line with the C timings produced better results. A new class for the 2812C was thus created. Note the comparison in timing in figure 18.

```
// WS2812 - 250ns, 625ns, 375ns
template <uint8_t DATA_PIN, EOrder RGB_ORDER = RGB>
class WS2812Controller800Khz : public ClocklessController<DATA_PIN, C_NS(250), C_NS(625), C_NS(375), RGB_ORDER> {};

// WS2812C - 300ns, 320ns, 470ns        - Done by me
template <uint8_t DATA_PIN, EOrder RGB_ORDER = RGB>
class WS2812C : public ClocklessController<DATA_PIN, C_NS(300), C_NS(320), C_NS(470), RGB_ORDER> {};
```

Figure 18, Custom LED Timing Class

While changing the timings was helpful, it still did not produce satisfactory results for the entire LED chain. It was initially thought that the operating system may be interrupting the transfer of the full chain of data. Whenever writing to the LEDs, 3 bytes for every LED must be sent out for each LED in the chain (up to the last LED being addressed). This means that 219 bytes must be written out, for the full chain. Testing with a logic analyser showed that this was note the case, however. This was not surprising in hindsight - the LEDs operate at 800KHz, so writing the full chain would still be well below the minimum 1ms time slice of FreeRTOS.

It appeared instead that the cause of the issue was the logic level analyser. The translation time of the logic level converter from 3.3V up to 5V was given as only 1ns (typical), but the signal integrity was evidently too degraded by the translation from 3.3V to 5V[12]. The trace from the ESP to the logic level converter was, in addition, nearly 17cm. For future revisions, the LEDs would have to be reassessed.

BPM timer

A high resolution timer was used to provide a BPM (beats-per-minute) interval for the sequencer. This system was composed of two parts - a timer task, that created the timer, and managed changes to it; and a timer ‘callback’, the ISR called when the timer fires.

```
void Timer(void *pvParameter)
{
    BPMclass BPMc;

    BPMc.BPMargs.callback = &BPMcallback; // Function called each time timer fires
            BPMc.BPMargs.arg = &BPMc;
            /* name is optional, but may help identify the timer when debugging */
            BPMc.BPMargs.name = "BPM";

    esp_timer_create(&BPMc.BPMargs, &BPMc.BPMtimer); // Create the timer with the given handle and arguments
    esp_timer_start_periodic(BPMc.BPMtimer, (60000000/BPM)); // 60000000 over BPM gives us BPM in us


    for(;;)
    {
    /*********Task Loop**********/
        if(BPMc.BPMchanged) // If BPM has changed
        {
            esp_timer_stop(BPMc.BPMtimer); // Need to stop timer before we can change period
            esp_timer_start_periodic(BPMc.BPMtimer, (60000000/BPM)); // Set new period based on new BPM
            BPMc.BPMchanged = 0;
        }
    taskYIELD();
    /*********Task Loop**********/
    }
}
```

Figure 19, BPM Timer Task

The timer task (figure 19) created another custom class, ‘BPMclass’, which stored the timer handle and configurations, as well as some local variables referenced against the main BPM variable. After creating the timer on startup, the task checks if the BPM has been changed. If so, it stops the timer, and starts it again with the new BPM - the timer must be stopped before its period can be changed.

```
// Callback - this is an interrupt that gets called whenever the BPM timer ticks
static void BPMcallback(void* arg)
{
    BPMclass* BPMC = (BPMclass*)arg; // This callback is given the address of the BPM object.
                                     // Just need to put it into a usable form

    if(BPMC->BPMp != BPM) // If the BPM has been changed
        {
            BPMC->BPMchanged = 1; // Reset the flag
        }
    BPMflag = !BPMflag;

}
```

Figure 20. BPM Timer Callback

The timer callback (figure 20) has two functions. It first checks if its local copy of the BPM is different from the actual (global) BPM. If so, sets the ‘BPMchanged’ variable to indicate to the timer task that the period should be updated. It also inverts a ‘BPMflag’ variable to indicate that the BPM has fired. This simple method was chosen to minimise the time spent in the timer callback - tasks that use the BPM do a similar operation on their end, by checking if their local flag variable is the same as the global to determine if a beat has passed.

Sequencer

With a BPM timer complete, it was possible to develop the sequencer. As with many other features, a class was created to group many related variables into a ‘track’ object. These tracks could then contain information such as their current active length, whether they sat on the top or bottom physical row of keys, what sort of MIDI message they sent, and which of their steps were currently set.

```
class TrackClass
{
    public:
    bool active; // Whether track is
    uint8_t tB;  // Which track bank
    bool topBot; // What position in
    uint8_t length = 15; // User conf
    uint8_t type; // Type of MIDI mes
    uint8_t d1; // Value 1 - Midi mes
    uint8_t d2; // Value 2 - Eg, for
    uint16_t steps = 0; // One bit fo
    uint8_t position; // Current posi

    void stepInc(uint32_t& Q3buff);
    void setStep();
    void velInc(bool direction);
};
```

Figure 21, Sequencer Track Class

The sequencer, through the ‘stepInc’ function, incremented the current position on the track each time the BPM callback indicated a beat had passed. If a step was active (meaning its corresponding bit in the ‘steps’ variable was set), then several operations were performed - the LED above or below that step would change colour, and the MIDI values stored in that track would be passed to the MIDI task, through a message queue.

Aside from the above mentioned MIDI difficulties with using both the keyboard and sequencer, this feature presented few difficulties in development. Potential avenues for future development in the sequencer will be discussed later in the report.

Keyboard

Like the sequencer, the keyboard presented few difficulties on its own. The small buttons on either side of the keyboard were used to shift an octave variable up and down. When a key is pressed on the keyboard, it is checked against the octave variable to determine the resulting note. The length of the keyboard is such that it can fit two full octaves. Though it somewhat increased the complexity, it was decided to allow the user to shift up and down octaves in an overlapping fashion, rather than in clean sets. For example, if the keyboard is currently set to octaves 1 & 2, pressing the octave up button will shift it to octaves 2 & 3, rather than 3 & 4.

As mentioned earlier, when problems were noticed in using both the keyboard and sequencer, the MIDI system was overhauled. In doing so, the keyboard was expanded to properly handle holding down keys, not just momentary key presses. In addition, the ability to adjust note velocity (for keyboard and sequencer) was also added.

```
if(i < 12) // Octave 0 area
{
    if(octave != 0) // If not in current octave (ie key cannot be held down for this note)
    {
        if((note[i].state == 1) | (note[i].state == 2)) // If note on was sent last round, or earlier
        {
            note[i].state = 3; // Send note off
        }
        else if(note[i].state == 3) // If note off was sent
        {
            note[i].state = 0; // Set to nil
        }
    }
}
```

Figure 22, MIDI Note Monitoring Excerpt

Screen

The screen for the Heron is a generic 320x240 pixel 3.5-inch ILI9341 screen with an XPT2046 touch panel on top. The UI was designed in C with the help of the LVGL set of libraries allowing for fast prototyping and task allocation.

The LCD display and touchscreen communicate with the ESP-32 via two SPI channels, one for each component. SPI was chosen over other protocols such as I2C due to the speed of transfer and the fact that the ESP-32 had two native channels available. All communication with the screen and touch panel was done through built in methods and libraries from Espressif allowing for the team to jump straight into putting things on the screen quickly.

UI design was important to get right due to it being crucial to the feedback from the system. Criteria for the screen feedback was such that information could be displayed in relevant chunks so the user wouldn't require too much physical intervention.

Criteria was laid out on what was wanted for the display and the team worked to meet them, with not all of the ideas making it in and new ones being added through feedback from testing.

- Changing any settings on the screen was low in importance as most would be faster to have the hardware on the board changing them such as the bank and track. The few things that were wanted to change using the screen was swapping the encoders and slider mode and saving and loading presets.
- Basic feedback should be shown in one place for easy reading, so values such as the BPM, Bank and Track among others are important to have in the same place.
- Seeing the progression of the sequence and its details as to which note is being used was a feature discussed but was scrapped due to deciding that the feedback from the LEDs on the board being good enough, but this feature could be revisited in the future
- Features were added after initial sketches such as having a preset saving and loading screen, and a visualiser screen was discussed, showing visual features such as a 2D variable plane, an equaliser, waveform, but in the end only the 2D plane made it in due to time constraints and the complexity of graphically creating those visuals.

Sketches were done to fit in some of the criteria, but features were scrapped or built upon to eventually reach the final layout. Referring to figure 1, some features such as the persistent button menu retained but persistent stats removed. The encoders have been retained but placed in a more friendly layout, but sequence editing and viewing as well as buttons for changing tracks and banks removed. The hardware was decided to be much more reliable when it came to changing things than the screen was.

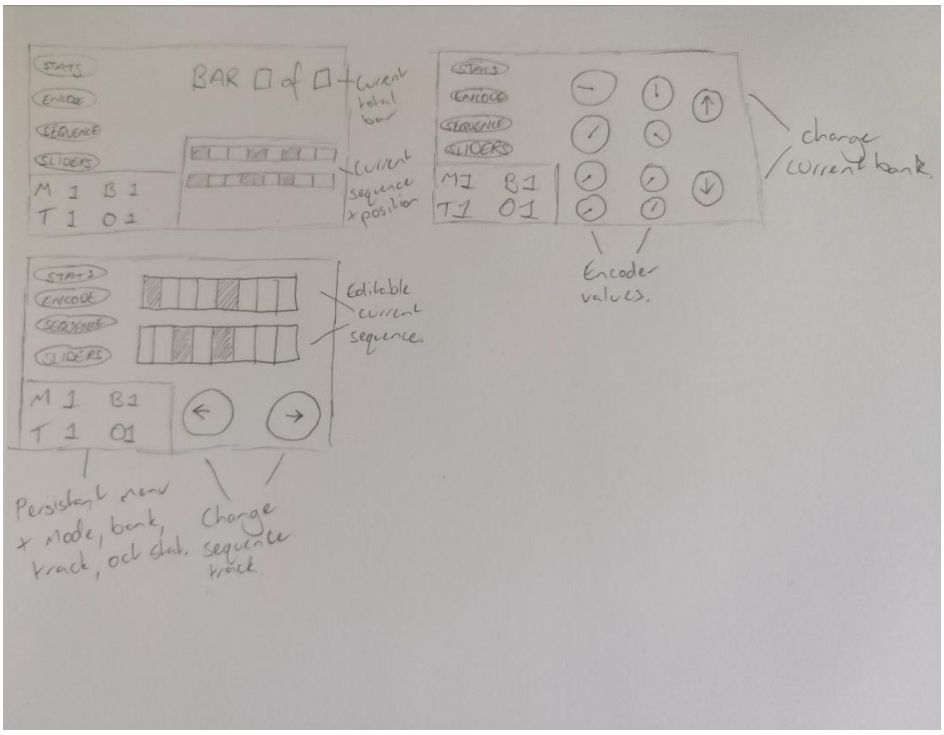
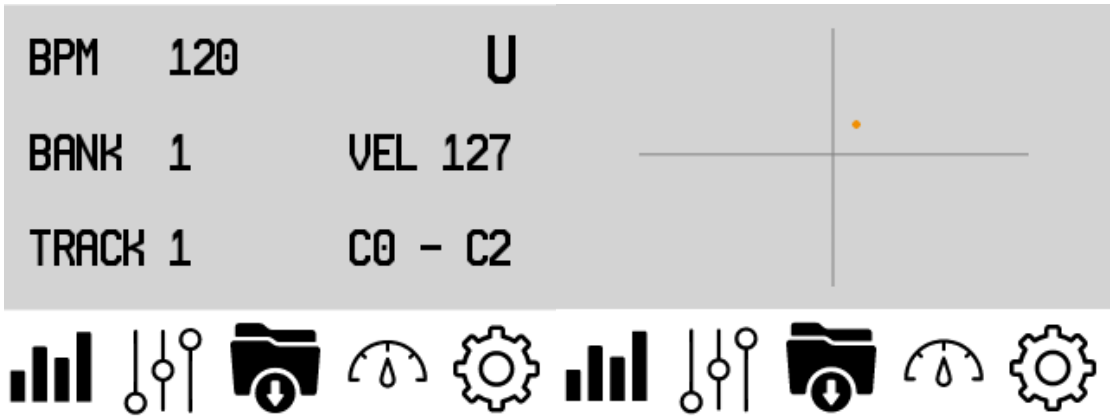
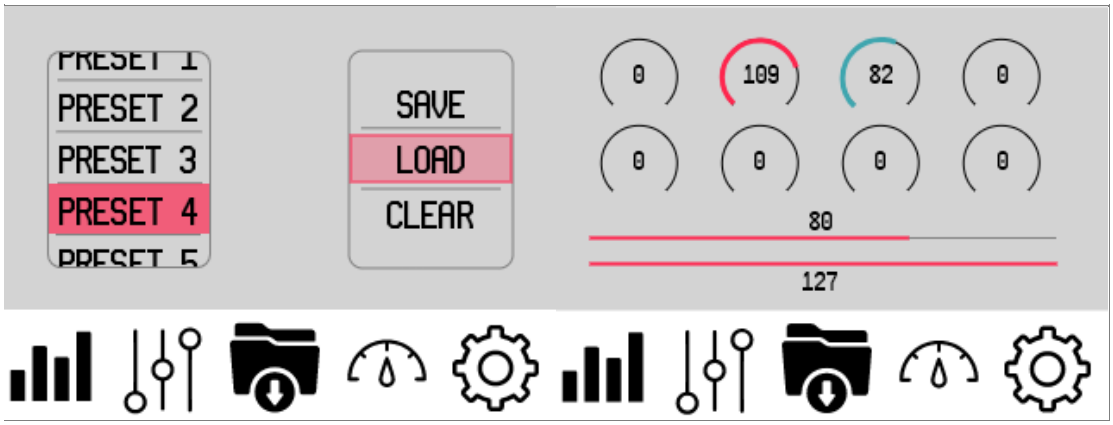


Figure 23 - Some early sketches showing some elements that were removed and some kept.

Eventually a set of five screens were chosen to contain all the features the team wanted to include. The screens met the criteria, providing all the feedback necessary to run the board, save the settings for another session later and quickly changing the encoder mode. It complements the code put into the running of the board well, and the integration of the screen and the board that were coded in isolation was easy, with the board code accessing variables in the screen to change the items on the screen. This is because screen animations and updating of all the text and movement was set up using tasks as built into the LVGL libraries that allow the user to code automatically updating animations that poll variables to see if the elements need to update. Conversely, changing variables using the screen's buttons affects the board's function such that when the encoder mode is changed, the encoders know to edit the right values.

The UI mock-ups were done in Lunacy which allows for fast design of UI elements, and the screen representations of these mock-ups are near 1:1 faithful recreations.





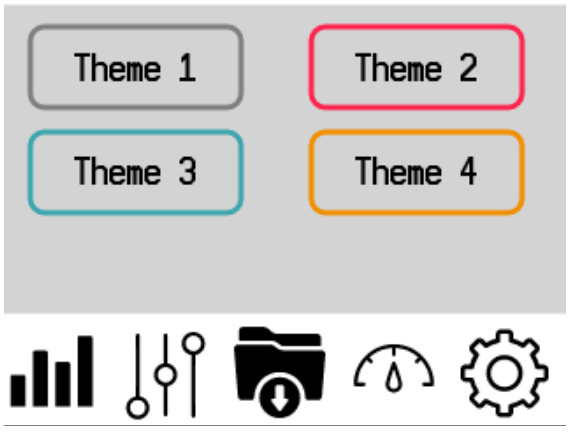


Figure 24 - Screens as included in the final version. Top left - Statistics screen. Top right - 2D variable screen. Middle left - Preset screen. Middle right - Encoder and slider feedback screen. Bottom - Theme settings screen.

Colours for the theme were chosen such that the arcs, bars, dots, symbols and text would all be clearly defined and legible on the background. A number of different themes using the same colours in different positions allow the user to pick one to their liking, with the colours being chosen for easy readability and aesthetics. The font used throughout the screen is NovaMono, a fixed width font for easy reading, aligning and fitting to elements like the encoder arcs and sliders. Across all screens is a persistent bar at the bottom with five symbols representing each screen, and tapping on the corresponding symbol will take the user to that screen. The symbols from left to right take you to the statistics screen, the 2D variable screen, the presets screen, the feedback screen and the settings screen.

Referring to the figure above, the statistics screen shows, as mentioned before, the BPM, bank, track, octave, velocity and mode, which is represented by a large letter for higher legibility. The letters displayed are U for utility, K for keyboard, S for sequencing, L for track length and N for note selection. All elements are evenly spaced from each other and the edge of the screen and values are vertically aligned so that they can be easily read no matter how many digits they have.

The 2D screen can display any two variables as if looking at their values on a 2D plane. This is especially useful for the joystick to modulate variables while playing.

The presets screen provides access to saving and loading presets for the board. Large buttons sitting on either side of the screen makes selecting the preset and choosing what to do easy. The preset list also scrolls allowing for more presets to be selected if needed. The presets persist through switching on and off the controller unlike the banks, so best used on finishing or starting a set.

The element feedback screen has eight arcs for the encoders, mimicking the layout of the physical encoders, and two bars for the analogue sliders. All elements start at 0 and max out at 127, and the current values are represented by a pink/red colour to contrast with the black bar and grey background. The encoder arcs can be pressed to enter CC mode for that encoder and turning that one now changes the effect that encoder is assigned to, with that value represented by a light blue. Pressing the symbol again goes back to the value changing mode and that effect can now be modulated using the relevant encoder. This is an important screen when setting up sounds, editing sequences, or keeping an eye on the effect values while playing in keyboard mode.

The theme settings screen contains four buttons that simply shift some of the colours around to the user’s preference.

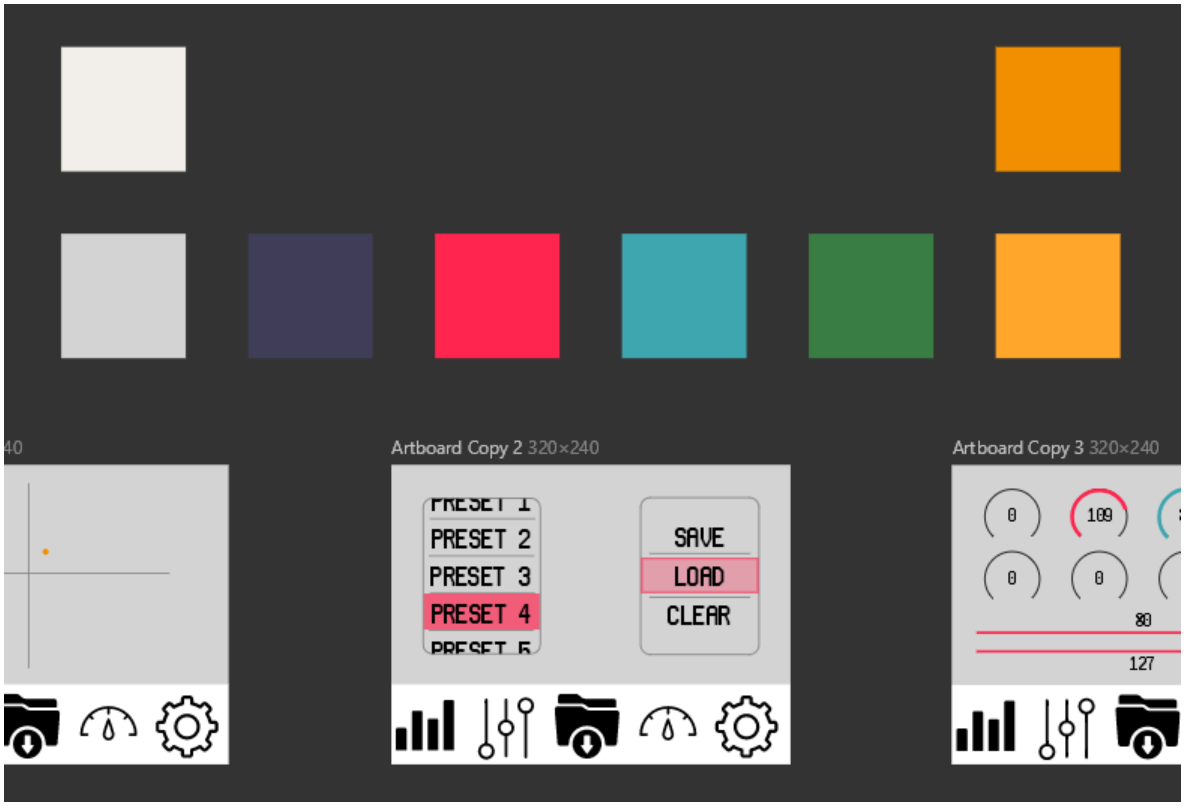


Figure 25 - Colour palette used for the screen. The colours changed slightly when actually putting the design on the screen due to the screen's colour capability but efforts were made to emulate as close as possible.

Components

The components used from the LVGL library for the UI are labels, buttons, a tabview, arcs, bars and lists. These components can be stylised to change their look to fit the users needs.

Labels

The labels are used for text, with the text being able to be updated through in built functions that put a user defined string into a character pointer list and push that into the set_text function. Note the bpm variable on the fourth line is the global BPM variable changeable by the board.

```
static void bpm_updater(lv_task_t * t)                    //BPM update task
  {
  static char buf[64];
  lv_snprintf(buf, sizeof(buf), "#000000 BPM   %d", bpm); //Label requires this format to be written to
  const char * texts[] = {buf};

  lv_label_set_recolor(bpm_label[0], true);               // Allows the text to be coloured
  lv_label_set_text(bpm_label[0], buf);                   // Sets the text
  }
```

Figure 26 - BPM label update task

The label is created here and is aligned to its space on the screen. The label's alignment values were slightly different to the UI design in Lunacy but that was due to margin differences in converting the font to C code that took a small amount of trial and error to overcome.

```
//BPM
bpm_label[0] = lv_label_create(parent, NULL);
lv_obj_align(bpm_label[0], NULL, LV_ALIGN_IN_TOP_LEFT, 10, 10);
lv_obj_add_style(bpm_label[0], LV_LABEL_PART_MAIN, &textBoxStyle);
```

Figure 27 - Label creation

To call the update task in figure 26 the code creates the task and readies it. Here the tasks all update at 100ms, which is quite fast but testing showed that the microcontroller was more than capable of running multiple tasks and it was felt that the more responsive the feedback, the better.

```
lv_task_t * t = lv_task_create(bpm_updater, 100, LV_TASK_PRIO_MID, NULL);       //BPM task
lv_task_ready(t);
lv_task_t * u = lv_task_create(bank_updater, 100, LV_TASK_PRIO_MID, NULL);      //Bank task
lv_task_ready(u);
lv_task_t * v = lv_task_create(track_updater, 100, LV_TASK_PRIO_MID, NULL);     //Track task
lv_task_ready(v);
lv_task_t * w = lv_task_create(mode_updater, 100, LV_TASK_PRIO_MID, NULL);      //Mode task
lv_task_ready(w);
lv_task_t * x = lv_task_create(octave_updater, 100, LV_TASK_PRIO_MID, NULL);    //Octave task
lv_task_ready(x);
lv_task_t * y = lv_task_create(velocity_updater, 100, LV_TASK_PRIO_MID, NULL); //Velocity task
lv_task_ready(y);
```

Figure 28 - Task allocation

Buttons

Buttons are used in the preset list and theme screen to begin an event.

The creation of the button is similar to the creation of the label. A position on the screen is given, but now a size can be set for the button and it is given an event to run on press.

```
//Button for testing
lv_obj_t * arc1_btn = lv_btn_create(parent, NULL);     //Add a button to the current screen
lv_obj_set_pos(arc1_btn, 20, 130);                          //Set its position
lv_obj_set_size(arc1_btn, 40, 40);                        //Set its size
lv_obj_set_event_cb(arc1_btn, arc1_btn_e);
lv_obj_add_style(arc1_btn, LV_STATE_DEFAULT, &listBtnStyle);
```

Figure 29 - Button creation

This button was used as a debug button and simply gives the arc and slider values a number to test the arc animations. On the second line it's shown if the action passed to the event was the button being pressed it would run the rest of the code. This means the button would toggle values allowing the team to try different alignments and colours quickly.

```
static void arc1_btn_e(lv_obj_t * obj, lv_event_t e){
  if (e == LV_EVENT_CLICKED){
    if (arcValues[0] == 110){
      arcValues[0] = 0;
    }
    else{
      arcValues[0] = 110;
    }

    if (arcValues[1] == 90){
      arcValues[1] = 0;
    }
    else{
      arcValues[1] = 90;
    }

    if (arcValues[2] == 127){
      arcValues[2] = 0;
    }
    else{
      arcValues[2] = 127;
    }
    if (slider1Val == 80){
      slider1Val = 0;
    }
    else{
      slider1Val = 80;
    }
    if (slider2Val == 20){
      slider2Val = 0;
    }
    else{
      slider2Val = 20;
    }

  }
}
```

Figure 30 - Button event example

Buttons are important to begin these events, and the most useful place it is used is in the lists for the preset saving and loading.

Tabview

The tabview is a feature that allows for multiple screens to be connected in one long horizontal row. It was hoped initially that the screen would be responsive enough to be able to swipe across screens but later testing showed that the resistive screen did have trouble registering accurate touches. Still, it offered a useful way of coding the UI as images could be placed over the top of a predefined five sections of the bar which would drag the display to that screen.

Figure 31 - Tabview with buttons

The tabview is created first in the script, and is given a background colour immediately.

```
lv_style_init(&bgStyle);
lv_style_set_bg_color(&bgStyle, LV_STATE_DEFAULT, lv_color_hex(0xF2EFEA));

tv = lv_tabview_create(lv_scr_act(), NULL);
lv_obj_add_style(tv, LV_PAGE_PART_BG, &bgStyle);
```

Figure 32 - Tabview creation

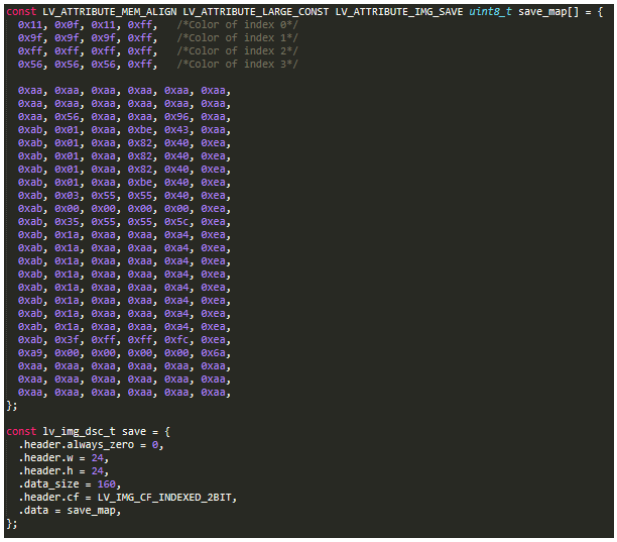
Tabs are added to the tabview which correspond to the number of screens sought on there. Next, image buttons are added and put on top. These buttons are identical to the other buttons but use images rather than the styles applicable to regular buttons.

```
t1 = lv_tabview_add_tab(tv, "Stats");
t2 = lv_tabview_add_tab(tv, "Visuals");
t3 = lv_tabview_add_tab(tv, "Presets");
t4 = lv_tabview_add_tab(tv, "Feedback");
t5 = lv_tabview_add_tab(tv, "Settings");

lv_obj_t * but1 = lv_imgbtn_create(lv_scr_act(), NULL);
lv_obj_t * but2 = lv_imgbtn_create(lv_scr_act(), NULL);
lv_obj_t * but3 = lv_imgbtn_create(lv_scr_act(), NULL);
lv_obj_t * but4 = lv_imgbtn_create(lv_scr_act(), NULL);
lv_obj_t * but5 = lv_imgbtn_create(lv_scr_act(), NULL);


LV_IMG_DECLARE(feedback55);
LV_IMG_DECLARE(sliders55);
LV_IMG_DECLARE(settings55);
LV_IMG_DECLARE(switch64);
LV_IMG_DECLARE(presets55);
LV_IMG_DECLARE(stats);
```

Figure 33 - Tabs and buttons added

At this point the five identically sized screens are created and allow for easy movement between them instead of clearing the screen and pushing a new screen on. All alignment of elements on each tab is done relative to that tab screen.

The images for the screen are kept as C code, converted from a png or jpg. The array gets drastically larger the larger the image and the more colours you want to include. For all the images there are four colours, a white background and three shades of grey.



```
const LV_ATTRIBUTE_MEM_ALIGN LV_ATTRIBUTE_LARGE_CONST LV_ATTRIBUTE_IMG_SAVE uint8_t save_map[] = {
  0x11, 0x0f, 0x11, 0xff,   /*Color of index 0*/
  0x9f, 0x9f, 0x9f, 0xff,   /*Color of index 1*/
  0xff, 0xff, 0xff, 0xff,   /*Color of index 2*/
  0x56, 0x56, 0x56, 0xff,   /*Color of index 3*/

  0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
  0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
  0xaa, 0x56, 0xaa, 0xaa, 0x96, 0xaa,
  0xab, 0x01, 0xaa, 0xbe, 0x43, 0xaa,
  0xab, 0x01, 0xaa, 0x82, 0x40, 0xea,
  0xab, 0x01, 0xaa, 0x82, 0x40, 0xea,
  0xab, 0x01, 0xaa, 0x82, 0x40, 0xea,
  0xab, 0x01, 0xaa, 0xbe, 0x40, 0xea,
  0xab, 0x03, 0x55, 0x55, 0x40, 0xea,
  0xab, 0x00, 0x00, 0x00, 0x00, 0xea,
  0xab, 0x35, 0x55, 0x55, 0x5c, 0xea,
  0xab, 0x1a, 0xaa, 0xaa, 0xa4, 0xea,
  0xab, 0x1a, 0xaa, 0xaa, 0xa4, 0xea,
  0xab, 0x1a, 0xaa, 0xaa, 0xa4, 0xea,
  0xab, 0x1a, 0xaa, 0xaa, 0xa4, 0xea,
  0xab, 0x1a, 0xaa, 0xaa, 0xa4, 0xea,
  0xab, 0x1a, 0xaa, 0xaa, 0xa4, 0xea,
  0xab, 0x1a, 0xaa, 0xaa, 0xa4, 0xea,
  0xab, 0x1a, 0xaa, 0xaa, 0xa4, 0xea,
  0xab, 0x3f, 0xff, 0xff, 0xfc, 0xea,
  0xa9, 0x00, 0x00, 0x00, 0x00, 0x6a,
  0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
  0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
  0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
};

const lv_img_dsc_t save = {
  .header.always_zero = 0,
  .header.w = 24,
  .header.h = 24,
  .data_size = 160,
  .header.cf = LV_IMG_CF_INDEXED_2BIT,
  .data = save_map,
};
```

Figure 34 - C code image for the Save icon in the presets menu. Code for the tabview images is much larger.

Arcs

The arc component is how the encoder values are displayed on the screen.

```
// ARC1
arc1 = lv_arc_create(parent, NULL);
lv_obj_set_drag_parent(arc1, false);
lv_arc_set_bg_angles(arc1, 0, 270);
lv_arc_set_rotation(arc1, 135);
lv_arc_set_angles(arc1, 0, 270);
lv_obj_set_size(arc1,  80, 80);
lv_arc_set_range(arc1, 0, 127);

lv_obj_align(arc1, NULL, LV_ALIGN_IN_TOP_LEFT, col1, row1);
lv_obj_add_style(arc1, LV_ARC_PART_BG, &arc_line);
lv_obj_add_style(arc1, LV_ARC_PART_INDIC, &arc_line_indic);

arcText[0] = lv_label_create(arc1, NULL);
lv_label_set_text(arcText[0], " 0 ");
lv_obj_align(arcText[0], NULL, LV_ALIGN_CENTER, -6, 0);
lv_obj_set_width(arcText[0], 50);
lv_obj_add_style(arcText[0], LV_STATE_DEFAULT, &arcTextStyle);
lv_label_set_align(arcText[0], LV_LABEL_ALIGN_CENTER);

lv_obj_set_event_cb(arc1, arc1_e);
```

Figure 35 - Arc creatiion and styling

The arc is created, rotated and modified such that its values run from 0 to 127 and it has a label in the centre of it aligned so that no matter the number of digits it will stay centred. It has two styles applied to it, one affecting the background of the arc such that the line consistently black, and one affecting the indicator, in the current case colouring it the UI palette red.

```
lv_style_init(&arc_line_indic);
lv_style_set_line_width(&arc_line_indic, LV_STATE_DEFAULT, 4);
lv_style_set_line_color(&arc_line_indic, LV_STATE_DEFAULT, LV_COLOR_MAKE(0xFF, 0x25, 0x4e));
lv_style_set_bg_opa(&arc_line_indic, LV_STATE_DEFAULT, LV_OPA_COVER);

lv_style_init(&arc_line_indic_cc);
lv_style_set_line_width(&arc_line_indic_cc, LV_STATE_DEFAULT, 4);
lv_style_set_line_color(&arc_line_indic_cc, LV_STATE_DEFAULT, LV_COLOR_MAKE(0x52, 0xd1, 0xdc));
lv_style_set_bg_opa(&arc_line_indic_cc, LV_STATE_DEFAULT, LV_OPA_COVER);
```

Figure 36 - The two styles applied to the indicator depending what state the arc is in.

The arc is also given an event on touch that will change what mode the encoder is in. Its default state is displaying the raw values in red, but on touching it will activate it's event, changing the mode, in this case indicated by "arcBool[0]" with the zero indicating the first encoder and so on up to 7 for the eighth encoder, and writing the new text, value and style to the arc.

```
static void arc1_e(lv_obj_t * obj, lv_event_t e){
  if (e == LV_EVENT_PRESSED){
    if (arcBool[0] == 0){
      arcBool[0] = 1;

      static char buf[64];      //Text writing to the label in the arc
      lv_snprintf(buf, sizeof(buf), "%d", arcCCVal[0]);
      const char * texts[] = {buf};

      lv_arc_set_value(arc1, arcCCVal[0]);  //Set the CC value
      lv_label_set_text(arcText[0], buf);
      lv_obj_align(arcText[0], NULL, LV_ALIGN_CENTER, 0, 0);  //Realign label to center
      lv_obj_reset_style_list(arc1, LV_ARC_PART_INDIC);       //Remove style
      lv_obj_add_style(arc1, LV_ARC_PART_INDIC, &arc_line_indic_cc);  //Add new style
    }
    else{
      arcBool[0] = 0;
      static char buf[64];  //Text writing to the label in the arc
      lv_snprintf(buf, sizeof(buf), "%d", arcValues[0]);
      const char * texts[] = {buf};

      lv_arc_set_value(arc1, arcValues[0]); //Set the CC value
      lv_label_set_text(arcText[0], buf);
      lv_obj_align(arcText[0], NULL, LV_ALIGN_CENTER, 0, 0);  //Realign label to center
      lv_obj_reset_style_list(arc1, LV_ARC_PART_INDIC); //Remove style
      lv_obj_add_style(arc1, LV_ARC_PART_INDIC, &arc_line_indic);//Add new style
    }
  }
}
```

Figure 37 - Changing encoder modes

When the encoder mode is "arcBool[0] = 0", the first encoder adjusts the raw values for the element it is changing on the synthesiser. When the encoder mode is "arcBool[0] = 1", it adjusts what element is going to be changed once that encoder goes back to default mode.

```
static void arc_anim(lv_task_t * t)
  {
    //Arc1
     if(arcBool[0] == 0){
       if (arcValues[0] == arcChecks[0]){  //Value of Arc1, checks if it is equal to it's previous value
       }                                   //and if it is, don't run the rest of the animation for this encoder
       else{
       static char buf[64];                                  //Text writing
       lv_snprintf(buf, sizeof(buf), "%d", arcValues[0]);
       const char * texts[] = {buf};

       lv_arc_set_value(arc1, arcValues[0]);                 //Set value
       lv_label_set_text(arcText[0], buf);                   //Set text
       lv_obj_align(arcText[0], NULL, LV_ALIGN_CENTER, 0, 0);  //Realign

       arcChecks[0] = arcValues[0];                          //Reset the previous value check so that the animation
       }                                                     //can check it again next time
     }else{
       if (arcCCVal[0] == arcCCChe[0]){ //Value of Arc1

       }
       else{
       static char buf[64];
       lv_snprintf(buf, sizeof(buf), "%d", arcCCVal[0]);
       const char * texts[] = {buf};

       lv_arc_set_value(arc1, arcCCVal[0]);
       lv_label_set_text(arcText[0], buf);
       lv_obj_align(arcText[0], NULL, LV_ALIGN_CENTER, 0, 0);

       arcCCChe[0] = arcCCVal[0];
       }
     }
```

Figure 38 - Arc animation

The arc animation code is quite long but this abstract in figure 38 shows how the first encoder works, which can be multiplied to all eight within this task. It first checks whether the current value is the same as the previous value, and if it is, it won't run the code. This is because if this check is not done, the board locks up as it tries to run the same commands repeatedly. Again, a text buffer is created and written to the label inside the arc, and the arc value is updated. The else statement does the same thing but for the blue CC values, running depending which mode that encoder is in.

Bar

The bar works very similarly to the arc, but it represents the linear sliders on the board. Unfortunately, the sliders were too small to touch to change mode like the arcs and since it was important to fit all the feedback on one screen it was necessary to change how that works. In the end, a button was hardcoded on the board to control what mode the sliders are in.

List

The list is essentially a scrollable page inside a page. The list can have buttons added to it which can be selected. The buttons are all exclusive so that two separate presets cannot be selected simultaneously, leading to misuse of the saving and loading features. The saving and loading functions are handled off the screen code so the save, load and clear buttons just set states for the board to interact with.

```
static void listPreset1_e(lv_obj_t * obj, lv_event_t e){
  if(e == LV_EVENT_CLICKED){
    printf("Event");
        if(lv_btn_get_state(listPreset2) == LV_BTN_STATE_CHECKED_RELEASED){ //If 2 is checked, uncheck
      lv_btn_set_state(listPreset2, LV_BTN_STATE_PRESSED);
      lv_btn_set_state(listPreset2, LV_BTN_STATE_RELEASED);
    }
        if(lv_btn_get_state(listPreset3) == LV_BTN_STATE_CHECKED_RELEASED){//If 3 is checked, uncheck
      lv_btn_set_state(listPreset3, LV_BTN_STATE_PRESSED);
      lv_btn_set_state(listPreset3, LV_BTN_STATE_RELEASED);
    }
        if(lv_btn_get_state(listPreset4) == LV_BTN_STATE_CHECKED_RELEASED){//If 4 is checked, uncheck
      lv_btn_set_state(listPreset4, LV_BTN_STATE_PRESSED);
      lv_btn_set_state(listPreset4, LV_BTN_STATE_RELEASED);
    }
        if(lv_btn_get_state(listPreset5) == LV_BTN_STATE_CHECKED_RELEASED){//If 5 is checked, uncheck
      lv_btn_set_state(listPreset5, LV_BTN_STATE_PRESSED);
      lv_btn_set_state(listPreset5, LV_BTN_STATE_RELEASED);
    }
        if(lv_btn_get_state(listPreset6) == LV_BTN_STATE_CHECKED_RELEASED){//If 6 is checked, uncheck
      lv_btn_set_state(listPreset6, LV_BTN_STATE_PRESSED);
      lv_btn_set_state(listPreset6, LV_BTN_STATE_RELEASED);
    }
        if(lv_btn_get_state(listPreset7) == LV_BTN_STATE_CHECKED_RELEASED){//If 7 is checked, uncheck
      lv_btn_set_state(listPreset7, LV_BTN_STATE_PRESSED);
      lv_btn_set_state(listPreset7, LV_BTN_STATE_RELEASED);
    }
        if(lv_btn_get_state(listPreset8) == LV_BTN_STATE_CHECKED_RELEASED){//If 8 is checked, uncheck
      lv_btn_set_state(listPreset8, LV_BTN_STATE_PRESSED);
      lv_btn_set_state(listPreset8, LV_BTN_STATE_RELEASED);
    }
    if(lv_btn_get_state(listPreset1) == LV_BTN_STATE_CHECKED_RELEASED){
      currBtn = 1;  //Set this variable to 1 so the save, load and clear functions know which preset to deal with
    }
  }
}
```

Figure 39 - Showing how the exclusive button states work

ADC

The analog components of the Heron interacted with the Axoloti in the same way as the encoders, but require a different method for reading their values. An ADC task was created to configure the ADC components, and obtain readings. The process required for reading the ADC on the ESP-32 is simple - after configuring the ADC channels used, raw readings can be taken. Generally, a number of samples are taken, then averaged out to provide a final reading - this is not strictly necessary, but is good practice.

ADC readings on the ESP-32 range from 12 to 9 bits in width. The readings taken were used to control MIDI CC values, which are 7-bits in width. For this reason, the lowest width, 9 bits, was chosen, as any higher readings were superfluous.

The readings proved to be accurate for most of the range, but the sliders did reach the top of the range in their readings slightly before reaching their physical maximum. This behaviour was not pronounced, but the ESP-IDF documentation [21] indicates that calibration functions are available that may improve accuracy, as individual units will have some variation in their reference voltages.

Preset Saving

Saving and loading presets on the Heron is achieved through the Non-Volatile Storage (NVS) system. The NVS system uses a portion of the flash memory to store data. The specific development boards used had 4mb of flash memory available, the vast majority of which was unused in the final product. The preset saving system was thus not overly constrained on space, as the partition tables for memory were easily modified to give a greater portion to the NVS.

It was considered highly desirable to allow users to save the CC numbers they had assigned to the encoders, sliders, and joystick. Without this ability, it would be necessary to remap a large number of variables each time the device was turned on.

Fortunately, the NVS system is not difficult to use. It requires only a single initialisation function, a function to open a storage area for operations, and a function to set or get a particular value in that storage. Figure 40 shows an example of the process during development.

```
nvs_flash_init();
nvs_handle storage_handle;
nvs_open("CC_Storage", NVS_READWRITE, &storage_handle);
nvs_set_u8(storage_handle, "Value1", var1);
nvs_get_u8(storage_handle, "Value2", &var2);
```

Figure 40, NVS Example

As discussed previously, eight presets were made available to the user through the touch screen. As each preset contains four banks of 12 elements (8 encoders, 2 sliders, 2 joystick axes), this means that 384 single-byte values were stored in the NVS. Thus, if a need was found to save other information, or future features required use of the NVS, there remains a great deal of space to work with.

RTOS

The various features discussed up to this point were linked together through the use of a real-time operating system. Espressif has developed a port of the FreeRTOS operating system for ESP-IDF. Among some other additions, the primary feature of this port is to enable multi-core support for dual-core versions of the ESP-32.

The dual core implementation of the operating system came with some warnings. Critical sections had some workarounds required from vanilla FreeRTOS, but these differences were not of importance for this project. Additionally, OS ticks are out of phase between the two cores (but otherwise in sync), meaning that they may not be acceptable for critical synchronization tasks. No situation was encountered during development were this was an issue, however.

The main concern was that, as explained in the ESP-IDF documentation [22], the priority list for tasks was shared between tasks pinned to separate cores. This could result in a situation where tasks are skipped, perhaps repeatedly, as one core increments the index in the list of ready tasks over a task that is pinned to the other core. For this reason, any tasks pinned to a single core were given a unique priority.

Ten separate tasks were created in the operating system:

Task	Core	Priority	Description
I2C Config	0/1	8	This task runs once at startup with a high priority, configuring the I2C peripherals used by other tasks.
UART Config	0/1	8	As above, for the UART peripheral.
Encoder Read	0/1	4	Reads the encoders through their IO expander.
Key Read	0/1	4	Reads the button matrix through its IO expander.
Variable Control	0/1	5	Manages tracked variables, like CC values or sequencer tracks. Takes input from most other tasks, and sends variable data out as needed.
Timer	0/1	2	Creates and manages the timer. The timer itself fires an interrupt, so this task does not need a high priority.
MIDI Send	0/1	5	Sends MIDI data from other tasks. Suspends itself when complete, and is resumed by other tasks as needed.

ADC Read	0/1	4	Reads analog components through ADC channels.
GUI Task	0	6	Displays a range of data on the screen, and takes input from touch controller. Some touch input passed on to other tasks. Pinned to one core by recommendation of library used.
LED Task	1	7	Displays data on LED chain. Suspends itself after LEDs are updated, and is woken by other tasks as needed. Pinned to one core as sharing a core with GUI task led to poor performance of LEDs or screen, even if LED task was permitted on both cores.

Some information was stored as global variables, such as the currently selected bank, or the current BPM. Global variables were primarily used to store information that was controlled in a single location, but accessed in many places. The primary means of communication between tasks, though, was through message queues. Seven queues were created:

No.	Msg. Size	Description
1	16 bit	Sequencer track queue. Used to send data from key task to variable control indicating when keys had been pressed in sequencer mode, to set or clear a step.
2	16 bit	Encoder task to variable control queue. As shown in figure 41, messages in this queue were broken into a number of segments to handle the variety of data that the encoders may be communicating.
3	32 bit	MIDI queue. A number of tasks used this to send data to the MIDI task to transmit. As shown earlier, the MIDI task filters out this data into three bytes to construct a MIDI message.
4	16 bit	Sequencer note queue. This queue sends data from key task to variable control to indicate that a new MIDI note has been selected for the current track.
5	64 bit	Encoders to screen queue. Sends the value of all encoders in the current bank to the screen to be displayed.
6	8 bit	Preset save queue. Only used to communicate preset save, load, and clear commands from GUI task to variable control.
7	32 bit	ADC queue. Sends data read from ADC to variable control to update slider and joystick values.

```
// Encoder received message structure:
// x/xxx/xxx/xx/xxxxxxx
// Check/Unused/Enc/Com/Value
// Commands - 00, write raw value
//          - 01, inc/decrement value (0/1 -/+)
//          - 10, change CC value
//          - 11, inc/decrement velocity (0/1 -/+)
// Top bit is set on any message (as an all 0 message would otherwise be valid)
```

Figure 41, Encoder Message Format

As features continued to be added to the project, performance decreased noticeably. This was surprising, in light of the considerable resources of the ESP (relative to the requirements). Testing seemed to suggest that the peripherals were the bottleneck in this system. The default tick interval for FreeRTOS is 10ms/100Hz, meaning a task will run for 10ms before yielding to another task, assuming the task does not yield itself, there are no interrupts, etc. Task yield instructions were placed throughout the code, but it became clear that these were not sufficient.

Take the I2C tasks as an example. The IO expanders used can communicate up to a speed of 1.7MHz, meaning that if the microcontroller waits for an I2C message to be sent before continuing with the program, it is massively limiting itself (the maximum speed of the ESP-32 used being 240MHz). It is not necessary for the microcontroller to do this, however, as the I2C peripheral is capable of handling much of the process on its own. The structure of the code written for the IO expanders, in line with the general process for communicating with those devices, involved several sequential I2C operations. The operating system was, therefore, sending the first message to the I2C peripheral, waiting for that message to conclude (and an acknowledgement to return), then sending the second message, and so on. Task yield commands were placed within the top level of the task, where they could only come before or after this entire process. Task yield commands could be inserted into the I2C process, between messages, but the best results came from increasing the tick rate to the maximum of 1ms/1KHz. After making this change, no more performance issues were encountered in the project.

Figure 42 shows the general relationship, and flow of information, from hardware devices to software tasks (and on to the Heron). As can be seen here, the variable control task became central to the entire system. While it was not necessary for all information to pass through here (for example, keyboard mode sent information straight from the key task to the MIDI task), data was generally sent to and from variable control.

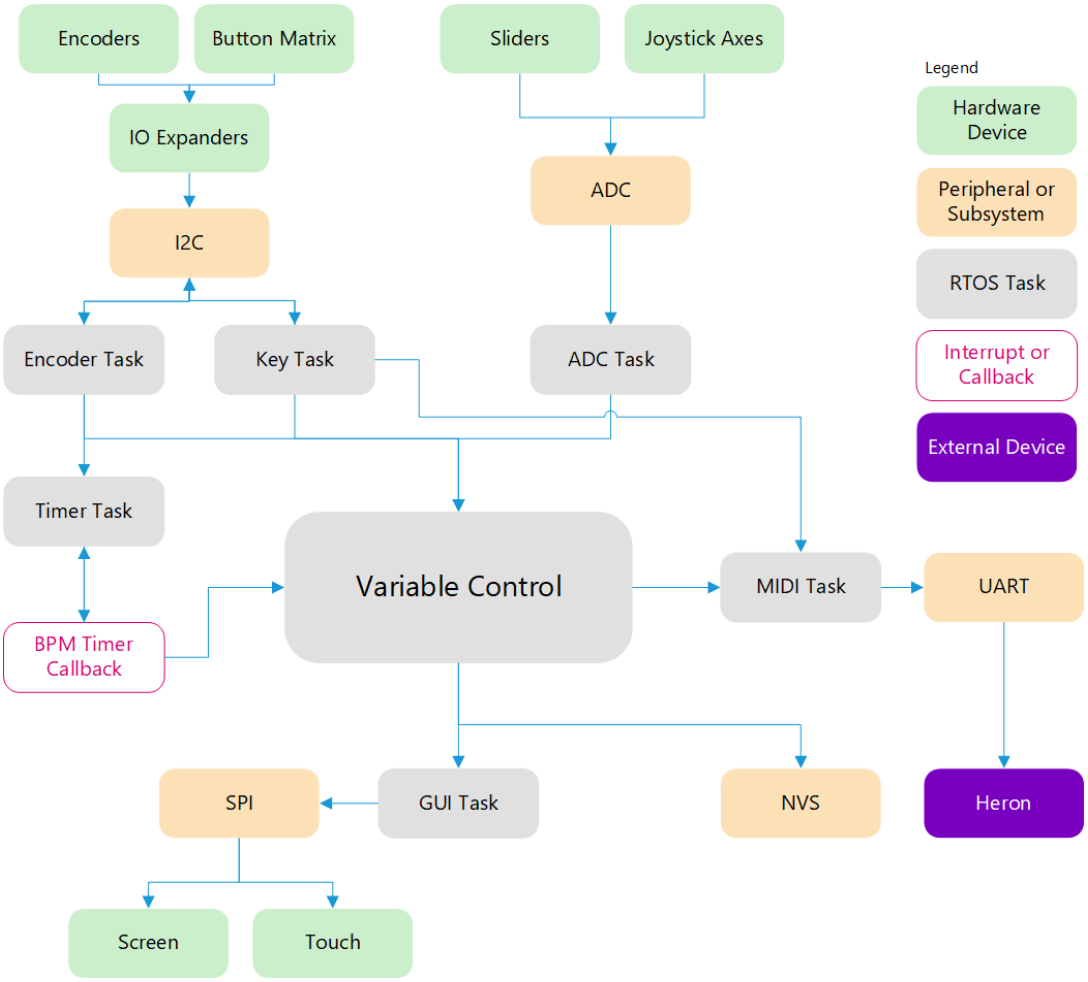


Figure 42, System Overview

If, for example, the encoder were turned, that data would (passing through the IO expander) be read by the Encoder task, using the I2C peripheral. If the device were in the appropriate mode to adjust the BPM, and the appropriate encoder (number 8) was being turned, this data would be passed on to the timer to update the BPM, which in turn notifies the variable control indirectly with a boolean flag when the BPM fires. If not, the encoder value would be passed to the variable control. The encoder task would save the previous encoder state, meaning specifically the state of the encoder pins, but is not concerned with the values it is controlling. The variable control task would then send the new value from the encoder turn both to the MIDI task, for transmission to the Heron, and to the GUI task for display on the screen. The GUI task would retain a regularly updated local copy of the data, and the MIDI task would send and discard the information as it makes its way through the MIDI queue.

As can be seen, the only interrupt used (aside from those underlying the operating system) was the timer callback for the BPM. If performance became an issue, the use of the interrupt pins on the IO expanders would be a good first step.

Progress Reviews

At various points during development, features were added, removed, or reassessed. These decisions were made by weighing the expected work required for each feature against the expected benefit to the final product.

Perhaps the most important decision made in this regard was the decision to use an operating system. The initial proposal for the project had the operating system as a potential method for organising the system, to be completed towards the end of development. It became clear, however, that this would not be a wise approach; the decision to use an operating system would have to be made early on in development, as it would shape how other tasks operated and interacted, and may require significant work on its own. Some initial investigation and testing was performed, as the group lacked any experience with real time operating systems, but it appeared that the use of an operating system may help, rather than hinder development. This was especially true given the number of distinct tasks involved; the use of an operating system would make it easier to add new features as OS tasks if and when they were developed. The use of FreeRTOS was an obvious choice for the project: aside from the permissive licensing, the existence of a specific dual-core port for the ESP-32 was key.

Wireless communication was considered, most likely the inclusion of bluetooth MIDI. The ESP-32, and ESP-IDF, is geared towards wireless/IoT applications, and the devboards used have antennas built in. Wireless MIDI would be desirable, but was not ultimately considered essential. Wireless MIDI is still far from ubiquitous, meaning that considerable work may well have been required to allow compatibility with a small number of devices. The option for including wireless MIDI remains open, however, for future development.

In line with this, consideration was given to developing a mobile app to accompany the Heron. A basic prototype for this app was developed in the Mobile Computing subject by a group member, and gives some indication of the intended functions (figure 43). This app would present a number of controllable elements that would act as the encoders or sliders did on the Heron board, and would also allow for controlling the sequencer on the mobile. While a prototype was developed that did send MIDI CC messages via bluetooth, development was not pursued further. The app was not seen as providing any functionality not already present on the board itself, making its utility dubious. Additionally, developing the app would necessitate the development of bluetooth for the Heron, which the team were already leaning away from.

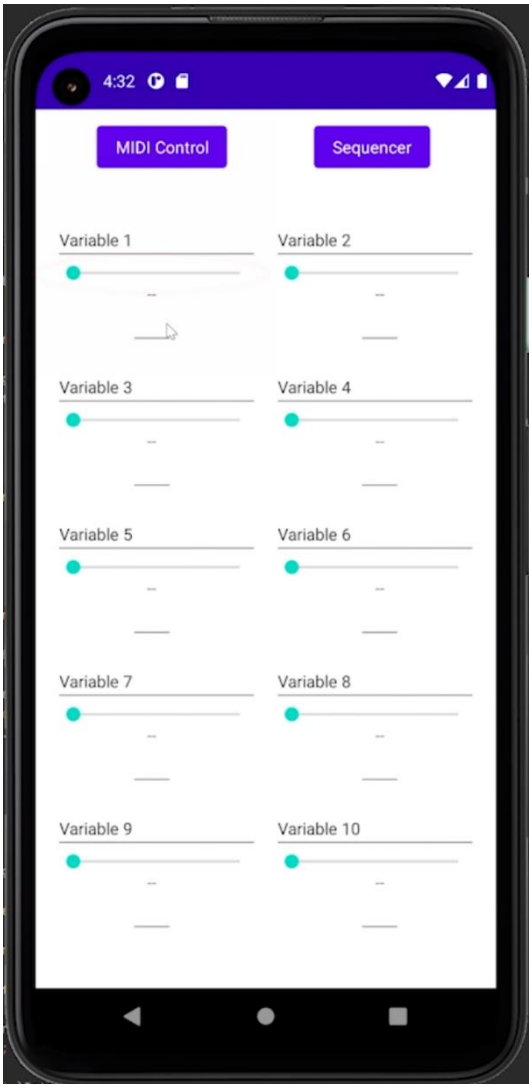


Figure 43, Heron Android App

Another factor in the decision not to pursue an app was the decision to include a relatively high-quality touch screen on the board. A 320x240 pixel TFT screen with a touch controller was chosen for the UI. Initially, some consideration was given to using a simpler, alphanumeric LCD screen. While this would be easier to develop, it was decided that attempting to present all the necessary information on such a screen would be less than ideal, and navigating the UI would be highly inconvenient without a more complex screen.

A number of external inputs and outputs were also planned. An analog input, most likely for expression pedals; MIDI in and MIDI out; clock in and clock out; and several miscellaneous digital outputs. These external connections would allow the Heron to expand beyond just interfacing with the Axoloti, and fit into a wider setup. This would also allow the Heron to bridge the gap between other devices and the Axoloti. MIDI input was primarily conceived of as a method for allowing the user to connect a keyboard to the Heron, if they did not wish to play on the keys provided. As the Axoloti has a MIDI input jack, this is not of vital importance, however. MIDI output, on the other hand, would allow the Heron to interact with other devices as it did the Axoloti, making it a more versatile device. The analog input would further increase the control options beyond the encoders, sliders, and joystick. The miscellaneous outputs were included largely because space was available, and it would be

easier to make use of them later if the need arose if jacks were already on the board. Unfortunately, a lack of time meant that these could not be implemented into the final project, limiting the Heron solely to interacting with the Axoloti. This is not a critical shortcoming, in that the main focus of the Heron is the Axoloti, but the added functionality of these extra connections would give the Heron greater flexibility, particularly for experienced users with more equipment.

Some thought was given to using an SD card for external storage. As the group had no experience with a project on this scale, the space requirements were unclear at first. The memory required for various UI elements, for example, could potentially be quite large. The screens purchased for the project (and many other generic screens) have SD card holders built in for this reason. Additionally, it was thought that an SD card could be used to save user presets on the board. None of this proved necessary, however. The non-volatile storage system was more than sufficient for preset saving, and the UI did not come close to filling up the memory available.

Finally, there were plans to write an ‘object’ for the Axoloti which would communicate information back to the Heron about the current patch. This could, for example, transmit the name of a variable controlled by a particular CC number, allowing the user to see this on the screen. These variables often do not have particularly informative names, however, meaning it would be more useful to allow users to simply write their own labels for a variable. Additionally, having text labels on the screen was found to make the layout more difficult to interpret at a glance, and was discarded in favour of numeric and graphical representation. Headers remain available on the board for more connections to the Axoloti should a use be found for them.

Results

At the conclusion of this project, the team is confident that they have produced a reliable product with a sufficient feature set for easy control of the Axoloti. This is achieved through hardware elements that control Axoloti variables through MIDI CC messages, a keyboard which the user can send MIDI Note On/Off messages to the Axoloti, and a sequencer which also sends Note On/Off messages in timed sequences arranged by the user. A touch screen and LEDs are present for presenting information to the user, and controlling the state of the Heron, and making assignments.

The encoders, sliders, and joystick axes act as intended on the Heron. The user may designate variables in an Axoloti patch to a certain CC number, and do the same for the Heron to link a patch variable to its hardware controller. By doing so, adjusting the hardware device (eg, turning the encoder) will generate a MIDI CC message that is sent to the Axoloti to control the associated variable. The encoders are the most basic elements here, with the sliders perhaps being more useful for variables whose entire range is likely to be used. The joystick is a somewhat less standard option, but provides an interesting method of controlling two variables relative to each other, or simply mapping one variable to an axis if you wish it to rest on its centre value.

As Axoloti patches can be exceptionally large, with many variables to control, four banks have been made available for the user. This allows the same hardware elements to control multiple CC numbers, based on the selected bank. While four banks (for a maximum total of 48 variables) would likely be more than enough for most users, this number could be expanded if testing showed it to be necessary. Related to this is the preset saving, loading, and clearing functions. It would be a very poor user experience to have to remap potentially dozens of variables each time the device is used. Furthermore, the Axoloti is designed for virtually endless combinations of objects to create its patches. Forcing the user to tediously reassign each time a new patch is used would run counter to one of the key benefits of the Axoloti. In light of this, eight slots have been provided for the user to save and load their CC assignments, either for different patches, or just different setups for the same patch.

The two rows of keys on the board have been laid out to allow for a two octave keyboard which the user can play. The keyboard can shift incrementally up and down octaves, has adjustable note velocity (as a standard aspect of MIDI note messages), and can handle multiple simultaneous key presses (as well as holding keys). While some users may well wish to use a full keyboard in their setup, for more casual use, or for beginners, this keyboard is likely sufficient. It allows the Heron to act as a capable standalone product.

Similarly, the keys on the board are used to provide a sequencer. The user may use the keys to set the track length, assign notes to the track, and de/activate steps on the sequence. The BPM of the sequencer is easily adjusted by the user, and the LEDs are used to indicate the position in the track, and the state of the steps on the track. As with banking on the control elements, the tracks have four banks of their own, each with two tracks on the two physical rows of keys.

The touch screen is a key part of the device, and conveys all the information deemed necessary in the Heron’s current state. The current mode of the board (keyboard, sequencer, etc), the BPM, selected track or bank, are all visible to the user through the screen. Another

tab on the screen displays the current values on the encoders and sliders, while also allowing the user to switch operations to select a CC number for these parts. Additionally, saving, loading, and clearing presets is performed through a tab on the screen.

The code written for the LEDs was functional, but hardware issues severely limited their operation. LEDs past (approximately) the 25th LED in the chain were not at all reliable, and LEDs before this point would sometimes have errors in displaying the wrong colours, or not updating. Where the LEDs did work mostly reliably, on the bottom row of keys, the system was felt to be very useful. The LEDs provided a vital way of providing the user with information at a glance, such as illuminating a light for which bank was currently selected, or which steps in a sequence were on or off. For the next version of the Heron, considerable work will need to be undertaken to provide a reliable hardware setup for controlling the LEDs.

Conclusions

The project to develop the Heron synthesiser control board aimed to provide a hardware and software solution for controlling the Axoloti synthesiser, which is a powerful and highly flexible device that lacks user interface. Various features were conceived, planned, reassessed, dropped, and completed throughout the year, and a stable product was delivered.

The features that would allow for interaction with devices other than the Axoloti, namely wireless communications and external device jacks, were not developed. It was felt that devoting time to these features would negatively impact the team's ability to deliver on the core feature set of controlling the Axoloti itself.

The Heron, as delivered, can control a large number of variables in an Axoloti patch through encoders, sliders, and a joystick. It can easily switch between presets for different patches, considered a key feature for usability. A basic, two octave keyboard is included for playing on an Axoloti patch. Finally, a multi-track, 16 step (adjustable) sequencer has been developed for arranging music on Axoloti patches.

Both a screen and LEDs are used to convey information to the user from the Heron. The screen is a relatively high resolution (240x320px) touch screen, which was developed successfully, and is capable of operating at a high refresh rate. The LEDs, on the other hand, were only partially functional. Errors in the hardware design of the board resulted in only a portion of the LEDs being usable, with reliability being an issue.

The use of the FreeRTOS real time operating system was vital in structuring the code in a manageable and efficient manner. The performance of the system is excellent, with no noticeable lag in any area. The use of an operating system will also provide a basis for future additions to the Heron.

The Heron is largely functional in its current state (with the above-mentioned LEDs being a notable exception). The system is fast and stable. While bugs no doubt exist in the code, they are not readily apparent. A number of features that were not attempted, or completed, during the year remain as possible avenues for future development, however.

Future Works

The primary recommendations for future work on the Heron are the planned features that were not developed, which have been mentioned previously in this report.

Wireless communications, particularly bluetooth MIDI, would be a welcome addition to the Heron. While not vital, as wired MIDI remains more prevalent than wireless, providing a wireless option would nonetheless provide compatibility with a wider range of devices, and may increase the longevity of the Heron as wireless MIDI becomes more common.

Related to this, the Heron mobile app would require a wireless connection. The app is not seen as an appealing addition to the Heron, however. The functionality it would provide would be limited at best. The Heron already possesses a touch screen, and the mobile interface for elements already on the board is unlikely to be an attractive alternative for most users.

In general, however, further external device functionality would be desirable. An analog input jack would require minimal extra development, and allow users to connect more control options to the board. MIDI input, as mentioned, is less important than a MIDI output jack. As MIDI is a widely used standard for musical devices, allowing external MIDI output would open up the Heron to a very large number of existing devices, as a keyboard, sequencer, and CC controller. Clock input and output may require a slightly more in-depth system to manage, especially input, but would allow the Heron to synchronize its BPM, either as master or slave, with other devices in an audio setup. Timing is obviously important to music, and as such this would be a useful addition. Finally, the miscellaneous outputs could potentially give the user the option of having configurable, generic outputs. These could, for example, be used as gate signals for Eurorack modules.

These additions, while not strictly necessary for the function of the Heron, may help to make it a more attractive product.

References

Background refs

[1] T. Holmes, *Electronic and experimental music*. New York: Routledge, 2008.

[2] R. Weidenaar, *Magic music from the telharmonium*. Metuchen, N.J.: Scarecrow Press, 1995.

[3] 1935 AEG Magnetophon Tape Recorder", *Web.archive.org*, 2021. [Online]. Available: https://web.archive.org/web/20130208162634/http://mixonline.com/TECnology-Hall-of-Fame/aeg-magnetophone-recorder-090106/. [Accessed: 18- Oct- 2021].

[4] C. Debussy, C. Ives and F. Busoni, *Three classics in the aesthetics of music*. New York: Dover, 1962.

[5] R. Worby, "WARP", *Warp.net*, 2021. [Online]. Available: https://warp.net/updates/richard-aphex-john-cage-and-the-prepared-piano. [Accessed: 18- Oct- 2021].

[5] "Time Out: “ I Want to be a Magnet for Tapes” ", *Music.hyperreal.org*, 2021. [Online]. Available: http://music.hyperreal.org/artists/brian_eno/interviews/timeo75a.html. [Accessed: 18- Oct- 2021].

[6] "SL MkIII | Novation", *Novationmusic.com*, 2021. [Online]. Available: https://novationmusic.com/en/keys/sl-mkiii. [Accessed: 18- Oct- 2021].

[7] "MPK Mini mkII", *Akaipro.com*, 2021. [Online]. Available: https://www.akaipro.com/mpk-mini-mkii. [Accessed: 18- Oct- 2021].

[8]"Launchpad X | Novation", *Novationmusic.com*, 2021. [Online]. Available: https://novationmusic.com/en/launch/launchpad-x. [Accessed: 18- Oct- 2021].

[9]"Learn more about Ableton Push", *Ableton.com*, 2021. [Online]. Available: https://www.ableton.com/en/push/. [Accessed: 18- Oct- 2021].

[10]M. Perrier, "Arturia - KeyStep - KeyStep", *Arturia.com*, 2021. [Online]. Available: https://www.arturia.com/keystep/overview. [Accessed: 18- Oct- 2021].

[11] Microchip Technology, “16-Bit I/O Expander with Serial Interface” , MCP23017/MCP23S17 Datasheet, June 2005 (Rev C, July 2016)

[12] Texas Instruments, “LSF010X 1/2/8 Channel Auto-Bidirectional Multi-Voltage Level Translator for Open Drain and Push-Pull Applications”, LSF01202 Datasheet, December 2013 (Rev J, May 2021)

[13] Texas Instruments, “SN74LVC1G14 Single Schmitt-Trigger Inverter”, SN74LVC1G14 Datasheet, April 1999 (Rev Y, November 2018)

[14] MIDI Manufacturers Association, “MIDI 1.0 Detailed Specification”, P. 1, February 1996, Los Angeles, California

[15]L. LLC, "LVGL - Light and Versatile Embedded Graphics Library", *LVGL*, 2021. [Online]. Available: https://lvgl.io/. [Accessed: 18- Oct- 2021].

[16]"Noun Project: Free Icons & Stock Photos for Everything", *Thenounproject.com*, 2021. [Online]. Available: https://thenounproject.com/. [Accessed: 18- Oct- 2021].

[17]Noun Project, " bar graph by Graphik Designz from the Noun Project", *Thenounproject.com*, 2021. [Online]. Available: https://thenounproject.com/. [Accessed: 18- Oct- 2021].

[18]Noun Project, " Download Folder by popcornarts from the Noun Project", *Thenounproject.com*, 2021. [Online]. Available: https://thenounproject.com/. [Accessed: 18- Oct- 2021].

[19]Noun Project, " Gear by Gregor Cresnar from the Noun Project", *Thenounproject.com*, 2021. [Online]. Available: https://thenounproject.com/. [Accessed: 18- Oct- 2021].

[20]Noun Project, " slider by Colourcreatype from the Noun Project", *Thenounproject.com*, 2021. [Online]. Available: https://thenounproject.com/. [Accessed: 18- Oct- 2021].

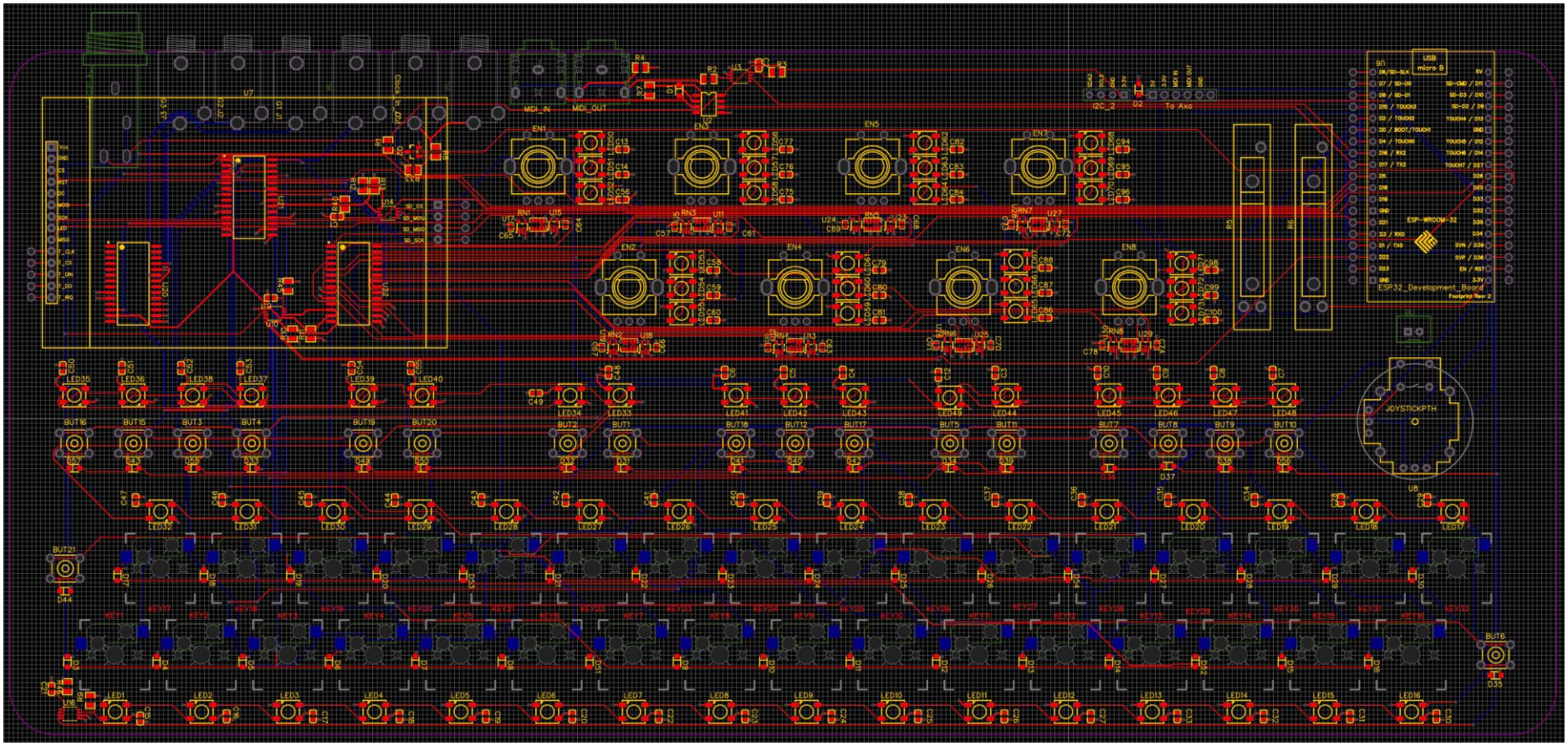
[21]Noun Project, " Gauge by Icons Bazaar from the Noun Project", *Thenounproject.com*, 2021. [Online]. Available: https://thenounproject.com/. [Accessed: 18- Oct- 2021].

[22]"espressif/esp-idf", *GitHub*, 2021. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/adc.html#adc-api-adc-calibration. [Accessed: 16- Oct- 2021].

[23]"espressif/esp-idf", *GitHub*, 2021. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/freertos-smp.html#round-robin-scheduling. [Accessed: 16- Oct- 2021].

Appendix

A - Final PCB Layout – Note that all components are present, but some connections were made or changed later in development.

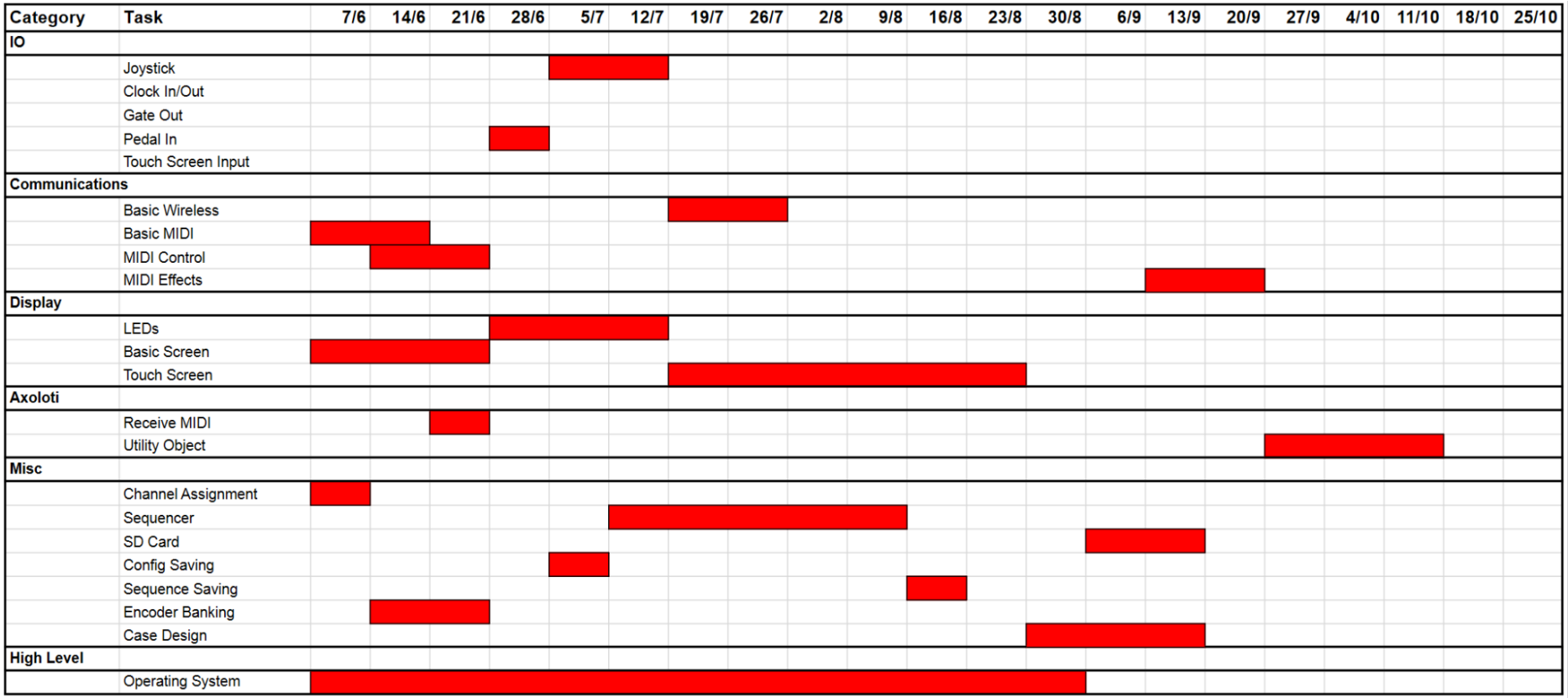


B - Bill of Materials

ID	Name	Footprint	Quantity	Manufacturer Part	Manufacturer	Price	JLCPCB Part Class	LCSC Assembly
1	Button	KEY-6.0*6.0-2	17	button6*6*8	ReliaPro	0.02		
2	100nF	C0603	77	CC0603KRX7R9BB104	YAGEO	0.0041	Basic Part	Yes
3	10nF	C0402	16	CL05B103KB5NNNC	SAMSUNG	0.0013	Basic Part	Yes
4	1N4148WS T4	SOD-323_L1.8-W1.3-LS2.5-RD	53	1N4148WS T4	CJ	0.0308	Basic Part	Yes
5	EC11E1834403	SW-TH_EC11E1820402	8	EC11E1834403	ALPS Electric	2.2271		
6	G1	PJ302M	1					
7	G2	PJ302M	1					
8	PEDAL1	AUDIO-TH_PJ-603	1	PJ-603	HOOYA	0.2794		
9	Clock_In_Out	PJ302M	2					
10	G3	PJ302M	2					
11	KEY16	KAILH_MX_SOCKET	1	Kailh				
12	Kailh Choc	KAILH_MX_SOCKET	31	Kailh				
13	WS2812C	LED-SMD_4P-L5.0-W5.0-BL	73	WS2812C	Worldsemi	0.1048	Extended Part	Yes
14	Header-Male-2.54_1x5	HDR-6X1/2.54	1	Header-Male-2.54_1x6	BOOMELE	0.0253		
15	I2C_2	HDR-4X1/2.54	1	Header-Female-2.54_1x4	Boom Precision Elec	0.0521		
16	MMBT3904-C20526	SOT-23-3_L2.9-W1.3-P1.90-LS2.4-BR	1	MMBT3904	CJ	0.0156	Basic Part	Yes
17	12K	R0805	2	0805W8F1202T5E	UniOhm	0.0029	Basic Part	Yes
18	240	R0805	3	0805W8F2400T5E	UniOhm	0.0029	Basic Part	Yes
19	200K	R0805	4	0805W8F2003T5E	UniOhm	0.0028	Basic Part	Yes
20	ALPS	RES-ADJ-TH_RS30111A602N	2	RS30111A9012	ALPS Electric	1.1127	Extended Part	Yes
21	270	R0805	5	0805W8F2700T5E	UniOhm	0.0029	Basic Part	Yes
22	1.2M	R0805	1	0805W8F1204T5E	UniOhm	0.0035	Basic Part	Yes
23	10K	RES-ARRAY-SMD_0603-8P-L3.2-W1.6-BL	8	4D03WGJ0103T5E	UniOhm	0.0074	Basic Part	Yes
24	MIDI_IN	SJ1-3523NG	1					
25	HCPL-0601-500E	SO-8_L4.9-W3.9-P1.27-LS5.9-BL	1	HCPL-0601-500E	AVAGO	0.8187	Basic Part	Yes
26	LSF0102DCTR	MSOP-8_L2.9-W2.8-P0.65-LS4.0-BR	4	LSF0102DCTR	TI	0.7069	Basic Part	Yes
27	MIDI_OUT	SJ1-3523NG	1					
28	SN74LVC1G14DCKR	SC-70-5_L2.1-W1.3-P0.65-LS2.1-BR	16	SN74LVC1G14DCKR	TI	0.0351	Extended Part	Yes
29	ESP32_Development_Board	ESP32 DEVELOPMENT BOARD V1	1	ESP32 Development Board				
30	3.2_TFT_LCD_SPI_TOUCH	ILI9341-3.2	1					
31	JST-XH-02	JST-XH-02-ROUND_PAD	1	JST-XH-02				
32	MCP23017-E/SO	SOIC-28_L18.1-W10.3-P1.27-LS10.3-BL	3	MCP23017-E/SO	MICROCHIP	2.604	Extended Part	Yes

C - Proposal Timeline

D – Progress Report Timeline



E – Completion Plan Timeline

Category	Task	1/8	8/8	15/8	22/8	29/8	5/9	12/9	19/9	26/9	3/10
Critical											
	Full MIDI System										
	UI Design & Implementation										
	Case Design & Manufacture										
	LED Programming										
	Virtual Banking & Channel Assignment										
	Configuration Saving										
	Sequencer										
	Slider Implementation										
	Pedal Input										
	Operating System Optimisation										
Non-Critical											
	Wireless Midi										
	MIDI Effects										
	I2C Communication to Axoloti