Pixie

Senior Design, Notre Dame Electrical Engineering, 2024

Abby Brown, Ethan Lau, Dani Nah, Alexa Zeese

1 Introduction

1.1 Sales Pitch

Picture this: you're a broke college student, it's 3 A.M., and the only thing you've written on your paper is your name. Your desk is a chaotic battlefield of notebooks, half-empty coffee cups, and procrastination-induced despair. Your dilemma has been brought about by your lack of experience juggling the numerous responsibilities dumped on you, which you now realize requires careful time management and prioritization – skills you have yet to develop. And the deadline? It's creeping closer with each passing second, and your motivation is on life support. But wait, despairing scholar, because salvation is at hand!

Introducing Pixie, the perfect desktop companion! Pixie is a pint-sized desk friend that combines both functionality and charm – helping you focus and manage your responsibilities while bringing some cheer to your study environment. Resembling a miniature computer, it brings the perfect touch of liveliness to bring back the pep in your step. Pixie can provide you with focus features such as study timers, note and reminder forwarding, wake/sleep display modes, and more! Pixie also brightens your study environment by displaying your favorite images and fun pixel art as backdrops against its focus tools. With Pixie, you'll be rescued from the clutches of your procrastination and quickly be on your way to mastering the art of productivity.

1.2 Problem Statement

Present-day technological devices boast unprecedented advancements, offering swift access to the latest news, information, and trends at the tap of a personal device's screen. While this newfound convenience has the potential to enhance and accelerate the work experience, college students often grapple with distractions and procrastination induced by these devices. Compounded by students' limited experience in managing real-world responsibilities, these technological advancements can become a significant source of distraction and procrastination. Moreover, uninspiring workplaces can exacerbate issues, leading to decreased motivation and productivity. These factors collectively make it challenging for a substantial number of students to effectively juggle multiple deadlines and tasks, often resorting to the common practice of pulling all-nighters to complete assignments.

1.3 Proposed High Level Functionality of Pixie

To help prevent the temptation of becoming sidetracked by a cell phone during study sessions, we propose a device designed to provide the user with necessary information independent from their phone. The Pixie Display aims to create a more conducive environment for students to work or study by providing focus tools commonly accessed via a smartphone but without any unnecessary and distracting connections to the IOT (social media, entertainment platforms, etc.). For example, in the midst of active study sessions, Pixie can serve as a valuable study timer. Many contemporary students employ diverse study techniques, such as the Pomodoro method, advocating specific time blocks dedicated to focused work, followed by a 5 to 10 minute break. Pixie will also showcase the date, time and weather - information students frequently use their phones to access and as a result become distracted from their work. Other features Pixie offers

will be to display/forward user notes recorded on an app/website (accessed while in class or on the go) to the display for viewing when the user returns, and streaming bluetooth music with phone-free control of playback. Lastly,Pixie's display serves as a smart desk light that automatically turns on and off based on the user's wake/sleep hours. The display will also be programmed to light up in response to sound queues, like clapping, providing visibility in the middle of the night without the need for reaching for a phone. This approach sets the stage for a productive day by promoting proper rest and discouraging late-night phone usage.

At its core, Pixie is a helpful study tool students can use at their desk to increase productivity and encourage good study habits. By removing the student from their phone, Pixie will solve the problem of distracted scholars. However, as students ourselves, we are aware of the challenges to overcome the urge to use one's phone when studying. To ensure use of Pixie trumps this urge, we will also implement various aesthetic elements into the Pixie's user interface to make it an attractive and frequently utilized desktop assistant. Pixie will integrate its study tools/features into various display modes which serve the dual purpose of showcasing pixel art and pleasant to view screensavers. The user can choose from a handful of previously downloaded images and pixel graphics to display on their screen during work hours.

1.4 Realized Functionality and High Level Design

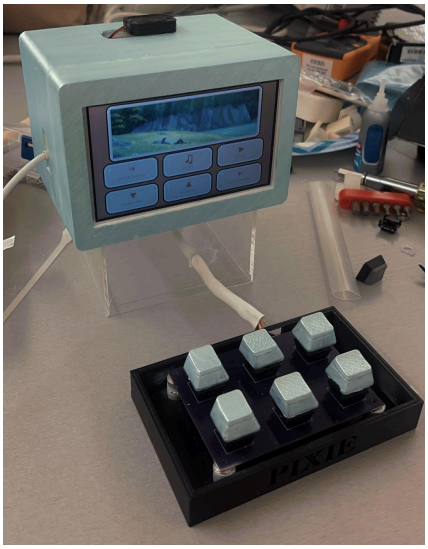
Over the course of the Spring 2024 semester, our team brainstormed, designed, tested, built, and integrated numerous hardware and software subsystems to create a working version of the Pixie Display. The user-facing components of our design include a 5 inch (800x480) LCD screen and a 6-switch keyboard. Using the keyboard as a physical interface, the user is able to interact with the embedded software to activate various hardware modules including Bluetooth, WiFi, flash memory chip, a speaker, and a microphone. The Bluetooth and WiFi modules are embedded in our chosen microcontroller, the ESP32-WROOM-32E. The LCD screen serves as the visual interface of Pixie, switching through various display modes depending on the keyboard input, providing feedback to indicate button presses, and of course, displaying fun graphics as screensavers and backdrops in the display modes.

At the outset of the project, our team laid out many features we wanted to incorporate into the Pixie display. The primary features we intended to implement were; location based weather/date/time information, timer functionality, bluetooth audio streaming, sound–activated night light, WiFi enabled notes/reminder forwarding, and display of fun pixel art/images selected by the user.

The goal of implementing this many features – which none of our members had any experience with – was incredibly ambitious. To add to the complexity of the project, our team is composed solely of electrical engineers who have limited experience in software development – a skill set necessary to smoothly integrate functionality of all our hardware components.

This said, our team worked incredibly hard and were successful in achieving the core design requirements set out for Pixie. We were able to successfully design, test, and develop all our peripheral hardware interactions/functions. This included: Bluetooth music streaming, WiFi enabled notes forwarding, WiFi enabled date/time/weather updates, keyboard inputs read and processed by the ESP32-WROOM-32E, display of text and graphical elements to the LCD screen, sound processing and illumination of the LCD screen, image uploading and reading from an external flash memory chip. After developing and proving the functionality of these subsystems we consolidated our hardware components onto a single PCB and integrated our software into one main program. This program controls the switching between display modes and activation of the aforementioned hardware functions based on user input from the keyboard. The only requirement not completely met in this fully integrated program is the sound activated illumination of the nightlight display mode. Although this hardware component worked flawlessly when tested independently, it caused a variety of issues in the task management of our software. We determined that in the interest of time and core functionality of Pixie that we would temporarily substitute the sound-activated night light feature for a Keyboard-input activation until we could further resolve the software issues. Although this is not what we initially envisioned for the nightlight feature, it still preserves the core functionality of Pixie and is a minor deviation from our initial requirements.

Pixie implements all the core features we set out to create, meeting our initial expectations (except for the sound activated night light). The keyboard provides a tactile way to interact with the various other hardware components/functionality while toggling through display modes. It is very accurate in displaying date, time, and weather as it uses the built-in WiFi on the microprocessor to obtain the real time and weather from the local WiFi and call to the openweather website via API. We are also very pleased with how well we were able to accomplish the visual and aesthetic elements of Pixie to make it an attractive alternative to using one's phone. The retro styled enclosure and display graphics resonate with Pixie's name and provide a pleasing to look at and use device.



Pixie: The image to the left shows the final product of Pixie as a whole with its user interface keyboard functionality and the stylish display screen & housing.

2 System Requirements

The requirements written for Pixie were determined based on the necessary tools students utilize while studying. Upon careful deliberation among the team and discussions with our peers, it was determined that students most often use the following study tools in their dorm room: bluetooth speakers/listening devices to provide study music, study timer, accessible note taking devices, low beam illumination devices, and a quick reference to the current date, time, and weather at the student's location. This list of common study features are found on students' personal cell phones. Thus, in order to achieve the product design goal of eliminating the need for a distracting cell phone while attempting to study, these study tools are designated as the leading features and requirements for the design of Pixie. Each feature is a subsystem within itself that obtains the necessary requirements to house all study tools into a single, unique study buddy. The following list of requirements were implemented from the beginning of this project. All listed requirements are fulfilled by the Pixie device and discussed in further detail under the Project Description section in this document.

2.1 Screen Display:

The screen of the device is the primary visual interface between the Pixie and its user. It shows pixelated art and widgets. It provides light, study timers, and date/time/weather data, acting as the primary connection between all subsystems. To display all of these features, a Liquid Crystal Display is used. The LCD interacts directly with the microcontroller, to display the user desired output.

Screen Type: LCD
Screen Size: 5" display
Features on Display:

- Settings screen: bluetooth pairing, setting timers, selecting pixelated art, etc.
- Show pixelated art
- Brightness turned on when user claps, activating night light feature
- Display date/time/weather
- Display activated timers
- Display specific synced notes

2.2 Push Buttons:

A mini keypad provides a local interface for the user to navigate and change the functionality of Pixie. This is where the user interacts with the screen display previously mentioned.

Push Button Type: Raised Key Cap Buttons
Total Buttons required: Six (6)
Connection to the main board/ESP32
Push Button purposes:

- Navigate to settings screen, arrow down/up to each settings feature, select settings
- Bluetooth pairing
- Set timers
- Select pixelated art
- Increase/decrease volume of speaker

2.3 Microphone:

A microphone is used to listen for sound cues. The auditory cues provide hands-free illumination of the display at night. Sound cues include clapping so that the user can speak to illuminate the night light screen display feature. The microphone is placed on the top of the display to obtain the best range for listening for the auditory cue.

2.4 Speaker:

The speaker provides audio to connect to the user's phone via bluetooth. The bluetooth playing feature is activated from a secondary screen shown on the display. Once bluetooth is connected, the user can adjust the speaker's volume and play/pause features from the Pixie Keypad and Display Screen. (No phones necessary!)

2.5 Study/Sleep Timer:

The device has the ability to set timers. Setting a timer for study purposes will enhance focus, allowing study break time to take place after the timer goes off. Furthermore, just like all timers, this feature may be used as an alarm for sleeping as well. Users can also choose from a default list of study methods (e.g., the Pomodoro method) that will automatically set the timers.

Internal clock: ESP32 timer functionality.

2.6 Notifications:

Another feature of the Pixie is that specific notes can be displayed on the Pixie screen.
The notes are connected to the display from the user's computer via a unique Pixie website and the WiFi module on the ESP32.

3 Project Description

3.1 System Theory of Operation

Pixie is controlled by the ESP32-WROOM-32E microcontroller which interfaces with the hardware peripherals in Figure XX (BLOCK DIAGRAM FIGURE NUMBER) to retrieve and send information. Functionality of Pixie is broken into a hierarchy of functions enabled by each hardware peripheral. The Display's function is responsible for executing the drawing of text and graphics. The Microphone's function is responsible for sending audio data to the ESP32-WROOM-32E. The Keypad's function is to send changes in button status to the ESP32-WROOM-32E. The Speaker's function is to play audio signals received from the ESP-WROOM-32E. The Flash's function is to write incoming data, or be available to have its data read by the ESP-WROOM-32E. The ESP32-WROOM-32E has the most complex function. At a high level, it is first responsible for the raw data processing for information received from any of the hardware peripherals. After processing this raw data, Pixie will execute the appropriate hardware functions described above. In a more detailed description, the ESP32-WROOM-32E accomplishes its function by implementing a Real Time Operating System (RTOS). This system uses time slicing and resource management to communicate and interact with the hardware peripherals in a seemingly parallel nature (described more below in Subsystem 6). Because the user input to control Pixie is sent through the Keyboard, Pixie's function focuses largely on monitoring the Keyboard input, debouncing and confirming "true" inputs. Based on which key was activated, Pixie will make resources available to the appropriate hardware peripheral and execute the hardware-specific functions described above.

3.2 System Block Diagram

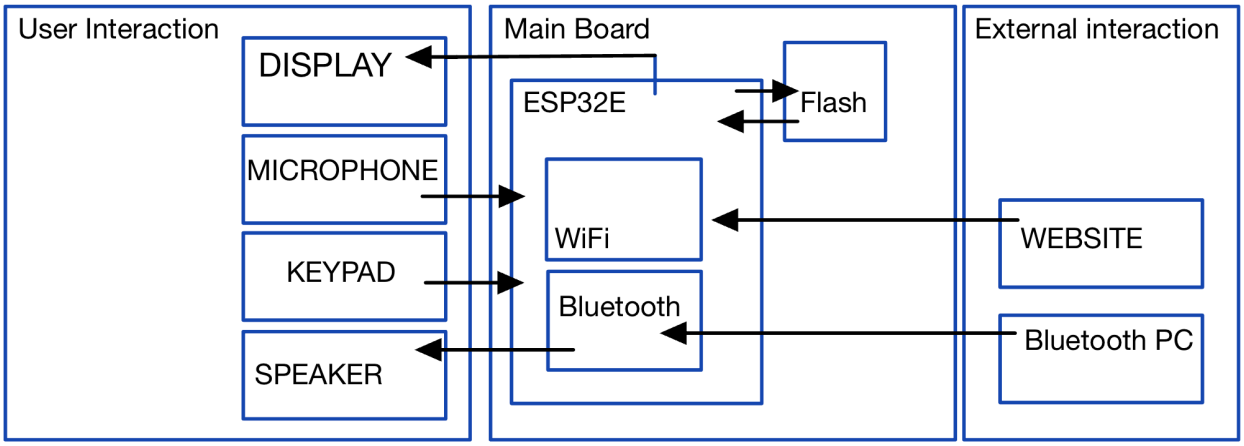


Figure 1: System Block Diagram

Shown above in Figure 1 is the System Block Diagram for Pixie. The central panel shows the ESP32-WROOM-32E microcontroller and an external Flash memory chip. The ESP32-WROOM-32E contains integrated WiFi and Bluetooth modules which communicate directly to the microprocessor. The Flash memory chip, used primarily to store BMP files (but can also store sounds for alarms, etc.) has bidirectional communication to the ESP32-WROOM-32E allowing the processor to both write and read data. The left panel shows the remaining hardware components necessary to achieve our requirements for Pixie. The Display communicates directly with the ESP32-WROOM-32E to receive draw commands to illuminate various pixel arrays to create our visual interface. The Microphone sends 32-bit digital to analog converted audio information to the ESP32-WROOM-32E for additional filtering and processing to extract information such as the relative dB level. The Keypad relays button pushes to the ESP32-WROOM-32E for debouncing and processing to toggle through display modes and activate hardware functionality. The Speaker receives digital information from the ESP32-WROOM-32E to output the analog audio information relayed via Bluetooth from the Bluetooth connected phone/PC, shown in the right panel. The right panel also shows the website and weather API communication to the ESP32-WROOM-32E's WiFi module, which is used to display date/time/weather and user inputted notes/reminders.

3.3 Subsystem 0 (Microcontroller) Design and Operation

Requirements:

The microcontroller subsystem is responsible for controlling all peripherals, delivering power to all peripherals, and programming the microcontroller. First and foremost, this subsystem must be capable of bridging USB to UART to program the microcontroller. Incoming 5V USB power must be regulated to 3.3V and capable of supplying 1.5A to the system. The microcontroller must provide SPI and I2S communication to control the display, flash chip, speaker, and microphone. An ADC must also be available to the speaker. Ample GPIO pins need to be available to connect all peripherals, including separate GPIO pins for each of the 6 keyboard switches.

Major Components:

Microcontroller: ESP32-WROOM-32E
The ESP32-WROOM-32E controller was chosen for its breadth of peripheral communications, processing capabilities, and adequate number of GPIO pins. With 32-bit dual core processing and 16MB onboard flash, the ESP32-WROOM-32E is capable of handling the software intensive control of peripherals via FreeRTOS task management. This microcontroller offers two SPI busses, two I2S ports, and 26 total available GPIO pins which ensures we will be able to make the necessary signal communications while not underutilizing a MCU with more GPIO pins. Additionally, the ESP32-WROOM-32E contains integrated Bluetooth and WiFi modules, providing ease of design and communication to these subsystems.

Programmer: CP2102
The CP2102 was selected for its capabilities to provide single chip integrated USB to UART data transfer without external resistors. The CP2102 provides USB 2.0 compliant data transfer and is compatible with both Mac and Windows operating systems, allowing all members of our team to establish a serial connection to the ESP32-WROOM-32E.

Voltage Regulator: AP7363-33D-13
The AP7363-33D-13 was selected for its voltage and current characteristics. This regulator is capable of supplying up to 1.5A with an ultra low dropout of 190mV at 3.3V. The typical quiescent current is also incredibly low at 0.5mA and changes minimally with loading. Our system demands a maximum of 1.3A with all peripherals (including WiFi) operating continuously. This regulator ensures we meet our power requirements, leaving ample headroom for unexpected surges/spikes.

Schematic:

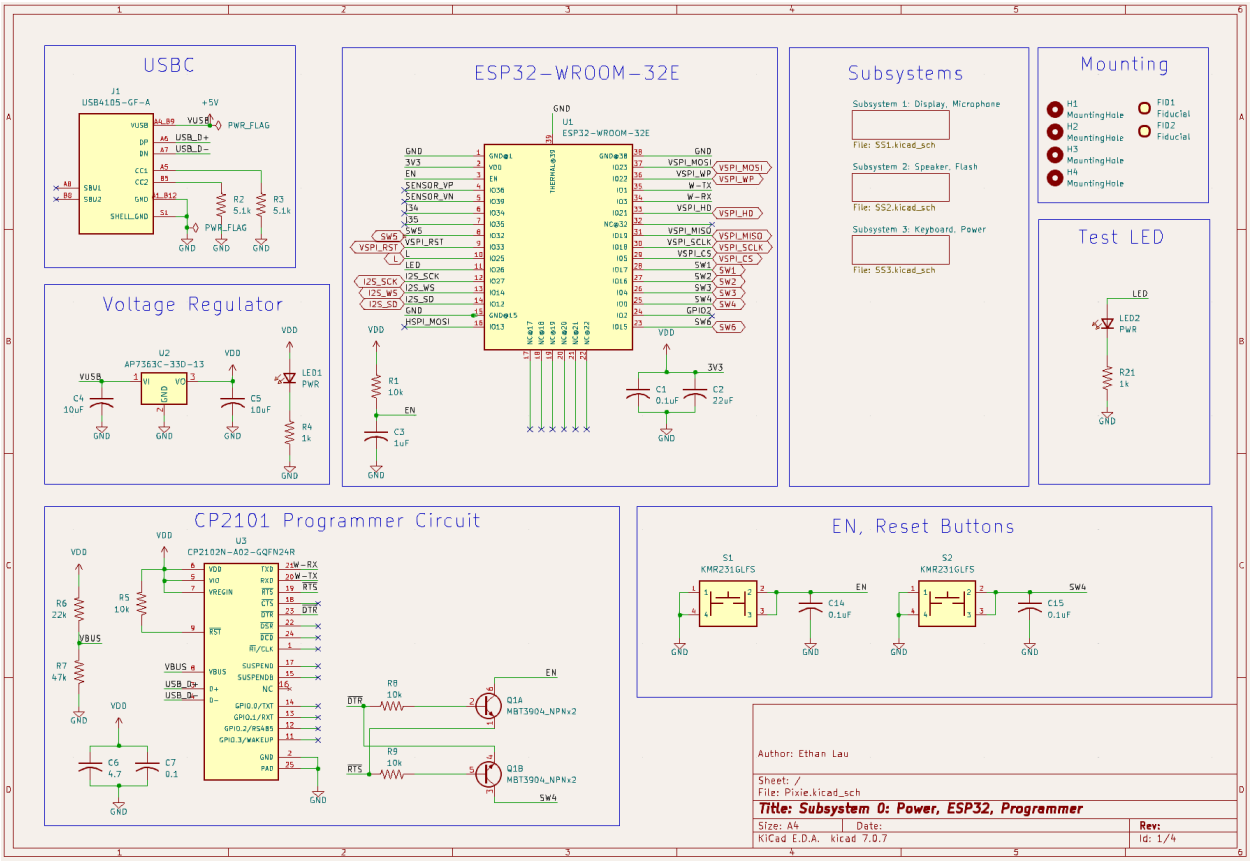


Figure 2: Schematic of Microcontroller Subsystem

In Figure 2, 5V power enters the system via the USB-C connector and is stepped down to 3.3V by the AP7363-33D-13 voltage regulator. This 3.3V supply is the voltage supply for all remaining components in this and subsequent subsystems. Differential data signals from USB-C are routed to the CP2102, which has VBUS biased to 2.25V to ensure the chip is configured to transfer the USB input. The output serial signals from the CP2102 W-RX and W-TX are routed to IO3 and IO1 of the ESP32-WROOM-32E, respectively. This connection establishes the serial connection to the microcontroller. The dual transistor connects the modem signals to enable programming. No pins require any strapping resistors except the EN pin, which is tied high to enable the chip on during power on. A push button grounding the EN pin to reset the chip is also connected to the EN pin. Another push button grounding GPIO0 is also connected to disable SPI boot. Listed below, by pin number (not GPIO number), are the allocations on the ESP32-WROOM-32E we have made to our peripheral systems. These pin configurations allow the ESP32-WROOM-32E to communicate and control all peripheral devices. Specifically, these include the SPI controlled Flash and Display, I2S controlled Microphone and Speaker, and GPIO

controlled keyboard. The Flash and Display subsystems share the same VSPI bus to reserve the other GPIO pins configurable for the HSPI bus to be reconfigured for other subsystems.

1. GND SYSTEM
2. 3V3 SYSTEM
3. EN SYSTEM
4. SENSOR_VP SYSTEM
5. SENSOR_VN SYSTEM
6. I34 SYSTEM
7. I35 SYSTEM
8. sw5 KEYBOARD
9. VSPI_RST DISPLAY FLASH
10. L SPEAKER
11. LED
12. I2S_SCK MICROPHONE
13. I2S_WS (REMOVED HSPI_SCLK) MICROPHONE
14. I2S_SD (REMOVED HSPI_MISO) MICROPHONE
15. GND SYSTEM
16. VSPI_CS1 FLASH
17. NC SYSTEM
18. NC SYSTEM
19. NC SYSTEM
20. NC SYSTEM
21. NC SYSTEM
22. NC SYSTEM
23. sw6 KEYBOARD
24. sw5 KEYBOARD
25. sw4 KEYBOARD
26. sw3 KEYBOARD
27. sw2 KEYBOARD
28. sw1 KEYBOARD
29. VSPI_CS DISPLAY
30. VSPI_SCLK DISPLAY FLASH
31. VSPI_MISO DISPLAY FLASH
32. NC SYSTEM
33. VSPI_HD FLASH
34. W-RX SYSTEM
35. W-TX SYSTEM
36. VSPI_WP FLASH
37. VSPI_MOSI DISPLAY FLASH
38. GND SYSTEM

3.3.1 Subsystem 0 Testing

Once our board was built and inspected for shorts, opens, and missing components, testing for the Microcontroller subsystem began. This entailed supplying 5V USB power to the board and monitoring the onboard power LED light. After observing the power LED illuminate, we received visual evidence that the voltage regulator was operating properly. We confirmed this by using a voltmeter to read 3.3V at the output of the regulator and across the LED/resistor network. With verification that the regulator was operational, we proceeded to test the CP2102 and ESP32-WROOM-32E. Using the serial port monitor on our Mac, we monitored whether or not the CP2102 enumerated on the USB-C port connected to the board. After observing the *cu.usbserial-1410* port appear after plugging in the board, we were able to confirm that the programmer circuit had successfully established a USB to UART bridge. Lastly, we tested the field functionality of the programmer circuit and the ESP32-WROOM-32E microcontroller by uploading a blink program to control the test LED connected to GPIO26. After successfully uploading the code and observing the blink, we confirmed that this subsystem was fully functional.

3.4 Subsystem 1 (Display and Microphone) Design and Operation

Requirements:

The Display and Microphone Subsystem is responsible for the visual interface between Pixie and the user and for auditory user input. Both the Display and Microphone must be capable of communicating to the ESP32-WROOM-32E for bidirectional data transmission. Specifically, the Display needs to communicate either via SPI or I2C, and the Microphone must communicate via I2S to satisfy the protocols available on the ESP32-WROOM-32E. The Display needs to be at least 5 inches measured diagonally and capable of color display. The Display should also have a backlight to satisfy its dual purpose as a nightlight. The microphone needs to be sensitive to sound levels between 40dB - 100dB to provide sufficient range for sensing auditory inputs. The microphone also needs to have a frequency response in the range covered by that of a clap (8kHz - 15kHz).

Major Components:

LCD Display: KD50G21-40NT-A1

The KD50G21-40NT-A1 display was chosen for its size, graphical versatility, and brightness. This display meets our minimum size requirements, having 5 inches of diagonal viewing area. The display is an 800x480 TFT LCD with a 24-bit controlled RGB screen. Because of its

physical construction, this display is capable of displaying 2^{24} possible colors, providing ample range for our system to display various graphics and viewing modes. The LCD has a powerful white LED backlight, providing 12 lamps of brightness; sufficient for dim night lighting required by our guidelines. The display is powered on 3.3V at 200mA (for max backlight brightness). Since this display is controlled via a 24-bit data line, additional hardware is required to control the display using the ESP32-WROOM-32E.

LCD Driver: RA8875

The RA8875 is an integrated LCD driver board. The RA8875 IC comes soldered with required peripheral devices on its own PCB to allow for SPI control of the 24-bit parallel data line. This driver board was chosen to drastically reduce the number of GPIO pins required to control the graphical display on the LCD (from 24 to 5). The RA8875 is powered on 3.3V. The RA8875 satisfies the requirements of our display subsystem by enabling full control of the LCD screen via SPI. We considered using the RA8875 IC directly on our PCB and connecting the supporting peripherals to enable SPI control, but decided to mount the pre-fabricated PCB on our Pixie PCB instead. This was because the pre-fabricated PCB had many passive devices and routes that would add density to our board and unnecessary complexity. Attempting to recreate this board on our PCB was possible, but increased the potential for errors and the need for debugging/rework.

Microphone: ICS-43434

The ICS-43434 is a MEMS sensor microphone package with embedded ADC, decimation, and anti-aliasing filters. This microphone communicates via 24-bit I2S and is powered from 3.3V at 1mA. This microphone was chosen primarily for its I2S communication protocol and its satisfaction for our sensing requirements. The ICS-43434 provides a wide frequency range between 60Hz - 20kHz and a sensing level up to 120dB with ±1dB accuracy. This microphone also includes an internal ADC, which will reduce signal processing on the microcontroller side and allow the ADC of the ESP32-WROOM-32E to be used exclusively for the speaker subsystem’s sound processing.

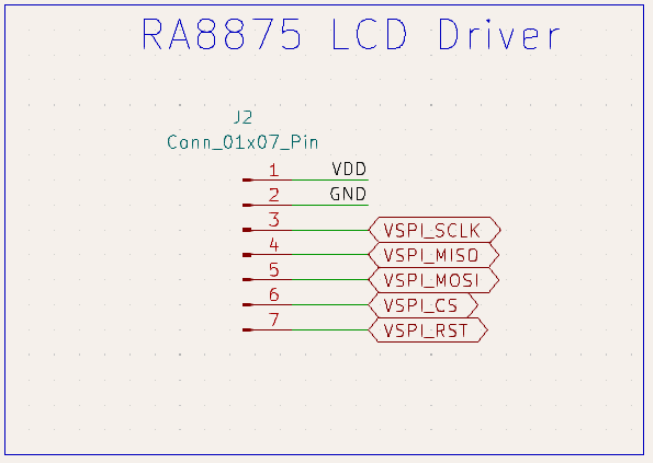
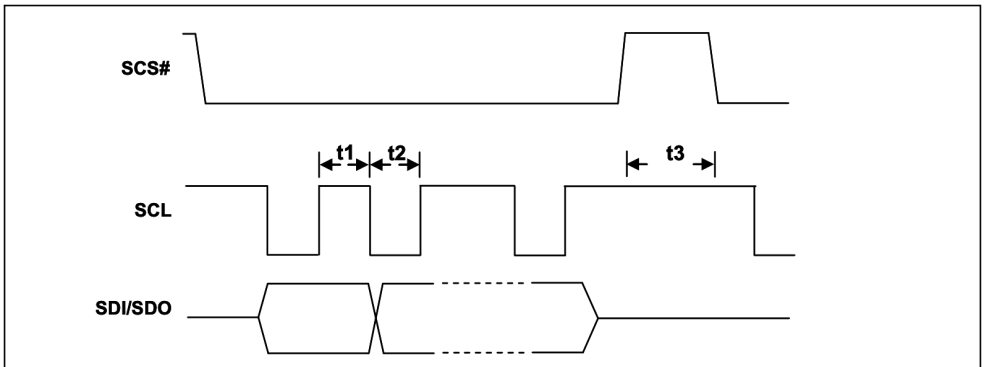


Figure 3: LCD Driver Schematic

Figure 3 shows the schematic for the LCD Driver. The LCD Driver is completely contained and serves as a SPI to 24-bit parallel bridge between the ESP32-WROOM-32E and the LCD Display. The ribbon connector of the LCD plugs directly into the RA8875 Driver. The Driver communicates to the ESP32-WROOM-32E via SPI. To allow for ease of positioning the Driver and LCD screen within Pixie's 3D printed enclosure, we chose to attach the driver board using ribbon/jumper cables which mount to a 1x7 vertical pin array. This pin array provides power and ground to the RA8875 and connects the SPI Clock, MISO, MOSI, Chip Select, and Reset signals to the pins defined in Subsystem 0.



Symbol	Description	Min.	Max.	Unit
SCL	SPI Clock frequency	100k	System clock/3(only write cycle) / System clock/6(with read cycle)	clock
t1	SCL High time	1.5 System Clock period	N/A	ns
t2	SCL LOW time	1.5 System Clock period	N/A	ns
t3	Minimum SCS# High Interval	5 System Clock period	N/A	ns

Figure 4: RA8875 4 Wire SPI Timing Diagram

Figure 4 shows the timing diagram used by the RA8875 to communicate to the ESP32-WROOM-32E via 4 Wire SPI interface. The slave device RA8875 is active low, indicated by the SCS# Chip Select signal. When held low, data on SDI (MOSI) and SDO (MISO) are valid and can be read on the rising and falling edges of the SCL clock line, respectively. Between data transfers, the Chip Select pin must be held high for a minimum of 5 System Clock periods before being asserted low to maintain data integrity. During data transfer (after SCS# pulled low), SDI will transfer commands to prepare the RA8875 to either receive or send data. The RA8875 will either listen, holding SDO low, or send the requested data.

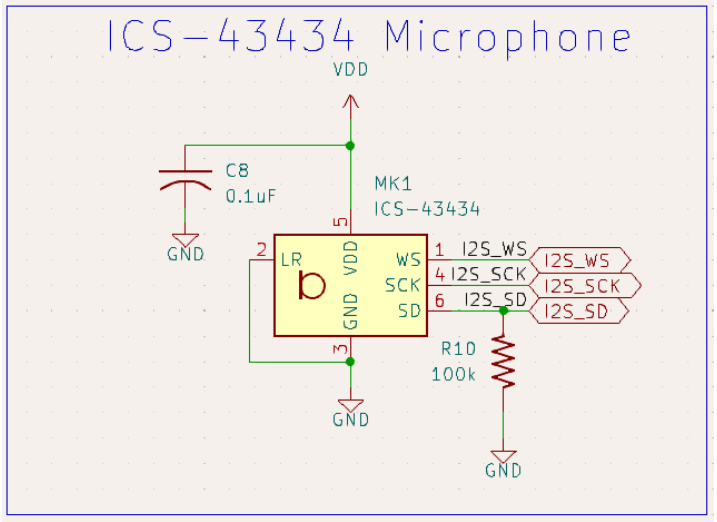


Figure 5: Microphone Schematic

Figure 5 shows the schematic for the Microphone. A decoupling capacitor of size 0.1uF (recommended by the datasheet) is attached to the power line of the microphone. A 100kOhm pull down resistor (recommended by the datasheet) is tied to the Serial Data line when no data is being sent/received. The Word Select and I2S Clock signals are connected to the pins defined in Subsystem 0. The LR signal is grounded to indicate a Left channel output of the microphone (this choice was arbitrary since only one microphone is being used. The LR signal could have alternatively been tied to VDD to indicate a Right channel output).

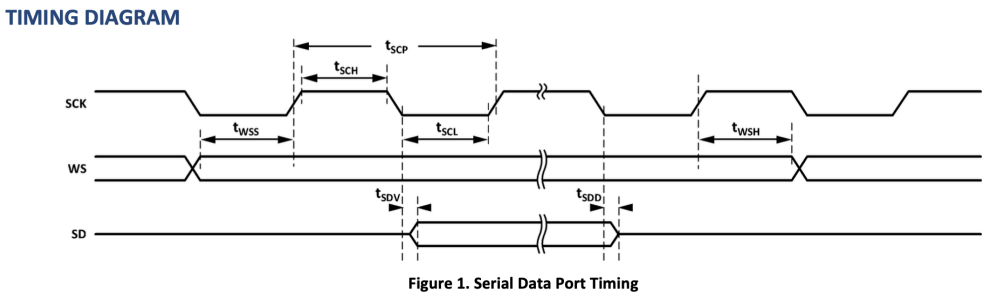


Figure 6: ICS-43434 I2S Timing Diagram

Figure 6 above shows the timing diagram used by the ICS-43434 microphone to communicate via I2S. SCK is the I2S clock signal. After setup and hold times, the WS line will change periodically at the start of each word. A low state indicates Left channel, and a high state indicates Right channel. The word data is sent on the SD line and changes on the falling edge of SCK so that it is stable to be read on the next rising edge. Even though the LR pin has been tied low to assert Left channel, WS must still toggle as part of the microphone protocol. Due to this, the data sent on the SD line will only be considered valid when WS is low (indicating Left channel).

3.4.1 Subsystem 1 Testing

Testing and verification of the Display and Microphone subsystems was accomplished in two parts; testing the functionality of the Display/Driver, and testing the functionality of the Microphone. Beginning with the Display/Driver, the appropriate power lines and SPI signals available via the pin header on our PCB were wired to the pin header on the Driver board using jumper cables. The ribbon cable of the LCD display was already attached to the Driver board from previous breadboard tests. An early version of our code that only interfaces with the Display/Driver to preview various display modes, which was also tested on a breadboard setup before having fabricated our custom PCB, was uploaded to the ESP32-WROOM-32E. Since the same pin assignments from our breadboard tests were used in our custom PCB, no changes to the software were made. Critical excerpts from this code are shown below in Figures 7 and 8, which show the definition of SPI pins and LCD Display object, and the setup of the LCD Display. Once these lines have been coded in definitions and setup portion of the code, common Adafruit graphical commands can be used to illuminate pixel patterns on the LCD Display. Upon uploading the code, the LCD display illuminated with the graphical displays designed in the prior weeks. This confirmed functionality of the Display/Driver connection.

```
19   // CS and RST assignments
20   #define RA8875_CS 5
21   #define RA8875_RESET 33
22   Adafruit_RA8875 tft = Adafruit_RA8875(RA8875_CS, RA8875_RESET);
23
```

Figure 7: Pin Definitions and Object Initialization

```
459     if (!tft.begin(RA8875_800x480)) {
460       Serial.println("RA8875 Not Found!");
461       while (1);
462     }
463
464     tft.setRotation(0);
465     tft.displayOn(true);
466     tft.GPIOX(true); // Enable TFT - display enable tied to GPIOX
467     tft.PWM1config(true, RA8875_PWM_CLK_DIV1024); // PWM output for backlight
468     tft.PWM1out(255);
469     tft.fillScreen(RA8875_BLACK);
470   }
```

Figure 8: LCD Display Setup Block

For the Microphone, we performed a similar test by uploading a program that interfaced exclusively with the ICS-43434. During the development phase of our project, we tested and developed our software for a similar microphone, the INMP441. The communication protocols between the INMP441 and ICS-43434 are identical. However, there are differences in the filter parameters and bits per sample configurations. We redefined the filter parameters to conform to the ICS-43434, and also redefined the bits per sample variable from 16 to 24. Changing the bits per sample from 16 to 24 also entailed increasing the storage variable from a uint16_t to a uint32_t, which needed to be changed throughout the remainder of the code. Another important change was the dB noise floor value, which was changed from 33 to 29 to match the specifications on the ICS-43434 data sheet. The microphone sensitivity parameters and filter definitions are shown in Figures 9 and 10, respectively.

```
40  //
41  // Configuration
42  //
43
44  #define LEQ_PERIOD        1           // second(s)
45  #define WEIGHTING         C_weighting // Also avaliable: 'C_weighting' or 'None' (Z_weighting)
46  #define LEQ_UNITS         "LAeq"      // customize based on above weighting used
47  #define DB_UNITS          "dBA"       // customize based on above weighting used
48  #define USE_DISPLAY       0
49
50  // NOTE: Some microphones require at least DC-Blocker filter
51  #define MIC_EQUALIZER     ICS43434//INMP441    // See below for defined IIR filters or set to 'None' to disable
52  #define MIC_OFFSET_DB     3.0103      // Default offset (sine-wave RMS vs. dBFS). Modify this value for linear calibration
53
54  // Customize these values from microphone datasheet
55  #define MIC_SENSITIVITY   -26         // dBFS value expected at MIC_REF_DB (Sensitivity value from datasheet)
56  #define MIC_REF_DB        94.0        // Value at which point sensitivity is specified in datasheet (dB)
57  #define MIC_OVERLOAD_DB   116.0       // dB - Acoustic overload point
58  #define MIC_NOISE_DB      29//33          // dB - Noise floor
59  #define MIC_BITS          24//16          // valid number of bits in I2S data
60  #define MIC_CONVERT(s)    (s >> (SAMPLE_BITS - MIC_BITS))//(s)
61  #define MIC_TIMING_SHIFT  0           // Set to one to fix MSB timing for some microphones, i.e. SPH0645LM4H-x
62
63  // Calculate reference amplitude value at compile time
64  const double MIC_REF_AMPL = pow(10, double(MIC_SENSITIVITY)/20) * ((1<<(MIC_BITS-1))-1);
65
66  // I2S pins
67  #define I2S_WS            14
68  #define I2S_SCK           27
69  #define I2S_SD            12
70
71
72  // I2S peripheral to use (0 or 1)
73  #define I2S_PORT          I2S_NUM_1
```

Figure 9: Microphone Sensitivity Parameters and Pin Definitions

```
77  // IIR Filters
78
79  // Definition of DC_BLOCKER filter coefficients
80  const SOS_Coefficients DC_BLOCKER_sos[] = {
81      {-1.0, 0.0, +0.9992, 0}
82  };
83  // Initialization of DC_BLOCKER using the template constructor
84  SOS_IIR_Filter DC_BLOCKER(1.0, DC_BLOCKER_sos);
85
86  // Definition of ICS43434 filter coefficients
87  const SOS_Coefficients ICS43434_sos[] = {
88      {+0.96986791463971267, 0.23515976355743193, -0.06681948004769928, -0.00111521990688128},
89      {-1.98905931743624453, 0.98908924206960169, +1.99755331853906037, -0.99755481510122113}
90  };
91  // Initialization of ICS43434 using the template constructor
92  SOS_IIR_Filter ICS43434(0.477326418836803, ICS43434_sos);
93
94  // Definition of A_weighting filter coefficients
95  const SOS_Coefficients A_weighting_sos[] = {
96      {-2.00026996133106, +1.00027056142719, -1.060868438509278, -0.163987445885926},
97      {+4.35912384203144, +3.09120265783884, +1.20841992636353, -0.273166998428332},
98      {-0.70930303489759, -0.29071868393580, +1.982242159753048, -0.982298594928989}
99  };
100 // Initialization of A_weighting using the template constructor
101 SOS_IIR_Filter A_weighting(0.169994948147430, A_weighting_sos);
102
103 // Definition of C_weighting filter coefficients
104 const SOS_Coefficients C_weighting_sos[] = {
105     {+1.4604385758204708, +0.5275070373815286, +1.9946144559930252, -0.9946217070140883},
106     {+0.2376222404939509, +0.0140411206016894, -1.3396585608422749, -0.4421457807694559},
107     {-2.0000000000000000, +1.0000000000000000, +0.3775800047420818, -0.0356365756680430}
108 };
109 // Initialization of C_weighting using the template constructor
110 SOS_IIR_Filter C_weighting(-0.491647169337140, C_weighting_sos);
```

Figure 10: Microphone Filter Definitions

Figure 11, below, shows the high level operation of the Microphone code from the setup block of the program. The code uses RTOS to pass samples recorded and processed by the mic_i2s_reader_task into the samples_queue. dB values are calculated based on the sample data. These dB calculations include over/underloading checks for the noise floor. The Leq_dB value is the average dB value over the Leq period of 1 second (line 44 of Figure XX). Once this value is calculated, the update_led_state() function is called which illuminates and LED (or the LCD Display) if the Leq_dB value exceeds a user defined threshold.

```
261 void setup() {
262   pinMode(LED_PIN, OUTPUT);
263   setCpuFrequencyMhz(80); // It should run as low as 80MHz
264   Serial.begin(112500);
265   delay(1000); // Safety
266   // Create FreeRTOS queue, task
267   samples_queue = xQueueCreate(8, sizeof(sum_queue_t));
268   xTaskCreate(mic_i2s_reader_task, "Mic I2S Reader", I2S_TASK_STACK, NULL, I2S_TASK_PRI, NULL);
269   sum_queue_t q;
270   uint32_t Leq_samples = 0;
271   double Leq_sum_sqr = 0;
272   double Leq_dB = 0;
273   // Read sum of samaples, calculated by 'i2s_reader_task'
274   while (xQueueReceive(samples_queue, &q, portMAX_DELAY)) {
275     // Calculate dB values relative to MIC_REF_AMPL and adjust for microphone reference
276     double short_RMS = sqrt(double(q.sum_sqr_SPL) / SAMPLES_SHORT);
277     double short_SPL_dB = MIC_OFFSET_DB + MIC_REF_DB + 20 * log10(short_RMS / MIC_REF_AMPL);
278     // In case of acoustic overload or below noise floor measurement, report infinty Leq value
279     if (short_SPL_dB > MIC_OVERLOAD_DB) {
280       Leq_sum_sqr = INFINITY;
281     } else if (isnan(short_SPL_dB) || (short_SPL_dB < MIC_NOISE_DB)) {
282       Leq_sum_sqr = -INFINITY;
283     }
284     // Accumulate Leq sum
285     Leq_sum_sqr += q.sum_sqr_weighted;
286     Leq_samples += SAMPLES_SHORT;
287     // When we gather enough samples, calculate new Leq value
288     if (Leq_samples >= SAMPLE_RATE * LEQ_PERIOD) {
289       double Leq_RMS = sqrt(Leq_sum_sqr / Leq_samples);
290       Leq_dB = MIC_OFFSET_DB + MIC_REF_DB + 20 * log10(Leq_RMS / MIC_REF_AMPL);
291       Leq_sum_sqr = 0;
292       Leq_samples = 0;
293       // Serial output, customize (or remove) as needed
294       Serial.printf("%.1f\n", Leq_dB);
295       update_led_state(Leq_dB);
296       Serial.printf("%u processing ticks\n", q.proc_ticks);
297     }
298   }
299 }
```

Figure 11: High Level Operation of Microphone Code

After the required changes to the code detailed above were made, the program was uploaded to the ESP32-WROOM-32E and we were able to successfully read the current dB sound levels via the serial monitor (see Figure 12) and activate an LED light by spiking the dB levels above 80dB (by clapping). With these observations, we were able to confirm full functionality of the ICS-43434 microphone.

Figure 12: dB levels measured by the ICS-43434 Microphone

3.5 Subsystem 2 (Flash) Design and Operation

Requirements:

The Flash subsystem is responsible for offloading the memory intensive task of storing BMP files from the ESP32-WROOM-32E. To draw images (rather than native graphics) using the RA8875 Driver, the color map of the image must be streamed to the Driver so that it can draw corresponding pixels on the LCD Display. BMP files are by nature uncompressed, and storing these files in SPIFFS consumes valuable CPU space for handling FreeRTOS tasks. Because of this, the Flash subsystem must provide an alternative storage location external to the ESP32-WROOM-32E memory. This external memory should be accessible using SPI communication and needs to be large enough (at least 64MB) to store a collection of full screen 800x480 BMP images (about 1-2MB each).

Major Components:

Flash Chip: W25Q128JVSIQ-ND
The W25Q128JVSIQ-ND is a 128MB flash memory chip accessed via SPI communication protocol. It supports standard, dual, and quad SPI for faster data transfers. This chip was chosen for its conformance to our requirements for SPI communication and minimum size requirements of 64MB.

Schematics:

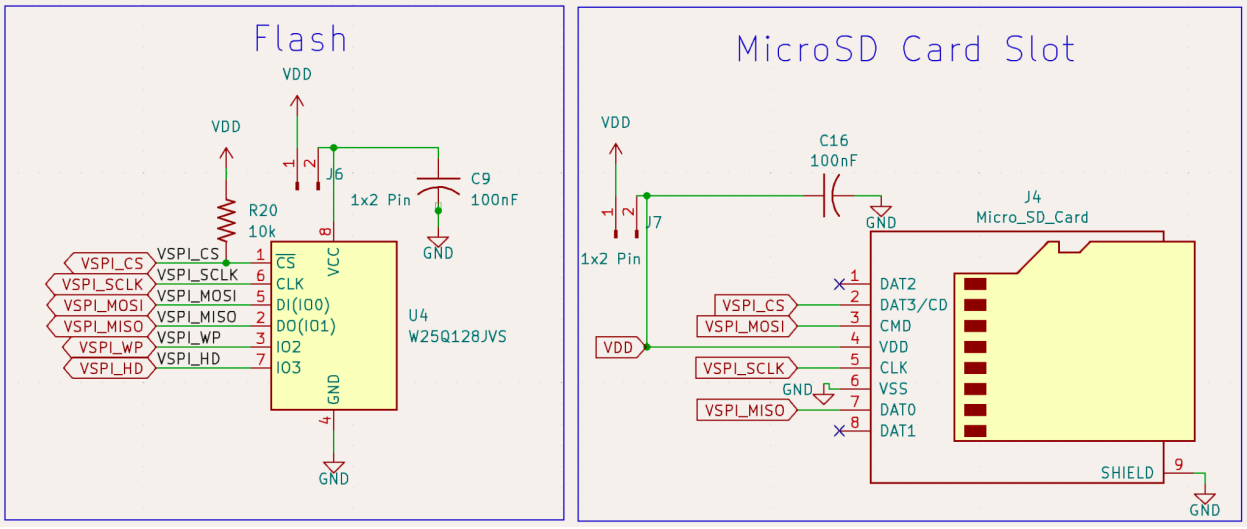


Figure 13: Flash Schematic

Figure 13 above shows the schematic for the Flash memory subsystem. In addition to the flash chip, an optional MicroSD card footprint was also added as an additional option for external storage. This was never utilized, but the option still exists for developers on our board. A jumper provides the power connection to either the Flash or the MicroSD, allowing selectability and power saving. The flash chip has two associated supporting passives. These are a power decoupling capacitor (100nF recommended by the datasheet) and a 10kOhm pull up resistor on the CS pin. This is to ensure the CS pin is normally high and the Flash chip is not occupying the SPI communication line to the microcontroller unless explicitly selected. Because the Flash shares the same SPI bus as the LCD Driver, the Flash CS pin is configured to GPIO13 instead of GPIO5. This ensures both the Flash and LCD Driver can be driven independently on the same SPI bus.

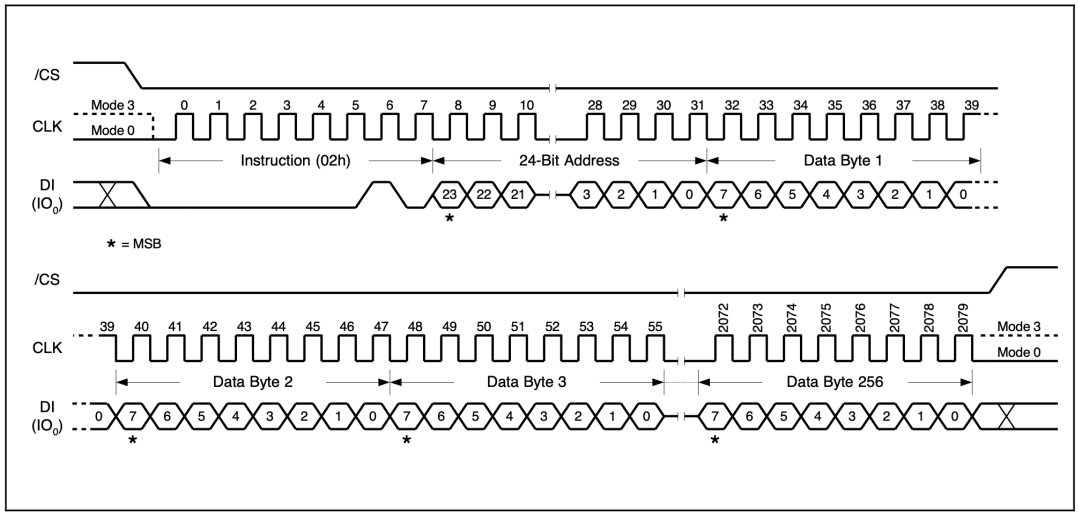


Figure 14: Page Program TIming Diagram

The SPI communication to the Flash chip is similar in principle to the SPI communication with the LCD Driver. However, since a hardware abstraction layer was not available for this flash chip, detailed hardware configurations were made at the command line. The timing diagram above shows one example of this hardware configuration for a page program command. First, the CS pin is driven low to select communication to the Flash chip. Then, an opcode (0x02h) is sent to the chip to prepare it for data reception and writing at the 24 bit address that follows. The actual sending of image (BMP) data is handled by a shell script located in the same directory on the PC that the BMP to be sent is located. To use this script, the ESP32-WROOM-32E is first prepared using a .cpp program that configures the Flash chip to receive data. In this .cpp program, the Flash chip can also be set to erase a 64KB, 128KB, 256KB, 512KB sector, or the entire chip, before programming. After uploading the code to the ESP32-WROOM-32E, the shell script is executed which byte packs the BMP data, sending it to the serial port of the ESP32-WROOM-32E to be written to the Flash chip at the specified address. This script sends data in little endian format, which is the format expected by the ESP32-WROOM-32E. Figure 15, below, shows the complete shell script. The script begins by defining the serial port to send data to, and the name of the BMP file to be transferred. The BMP file size is then calculated and rearranged in little endian format. The serial port is opened, and the file size is sent. Then, the script enters a loop that continues sending the remainder of the BMP data 256 bytes at a time. The script sleeps 0.1 seconds between transmissions to ensure the ESP32-WROOM-32E has adequate time to process and write the data to the flash.

```
#!/bin/bash
# Set the serial port and the BMP file path
SERIAL_PORT="/dev/cu.usbserial-1410"
BMP_FILE="music.bmp"

# Get the file size and convert it to a little-endian hex string
FILE_SIZE=$(stat -f%z "$BMP_FILE")
FILE_SIZE_HEX=$(printf "%08x" $FILE_SIZE | sed 's/\(..\)\(..\)\(..\)\(..\)/\4\3\2\1/')

# Open the serial port at 115200 baud
stty -f $SERIAL_PORT 115200

# Send the file size in binary format, directly using printf with the hexadecimal value interpreted as binary data
printf "\x${FILE_SIZE_HEX:0:2}\x${FILE_SIZE_HEX:2:2}\x${FILE_SIZE_HEX:4:2}\x${FILE_SIZE_HEX:6:2}" > $SERIAL_PORT

# Send the BMP file in chunks of 256 bytes
CHUNK_SIZE=256
for ((i = 0; i < $FILE_SIZE; i += $CHUNK_SIZE)); do
    dd if="$BMP_FILE" bs=1 skip=$i count=$CHUNK_SIZE 2>/dev/null | cat > $SERIAL_PORT
    # Adjust the sleep duration as needed to ensure ESP32 processing time
    # Correctly calculate bytes sent for the final chunk
    bytes_sent=$((i + CHUNK_SIZE))
    sleep 0.1
done
echo "Bytes sent: $bytes_sent"

echo "File transfer complete."
```

Figure 15: Byte Packing Shell Script

3.5.1 Subsystem 2 Testing

Testing and verification of the Flash chip was accomplished by uploading the platform.io .cpp program to the ESP32-WROOM-32E to prepare the Flash chip for programming. In this .cpp program, there are safety measures to ensure the Flash chip is enumerated and ready to write. An example of one safety measure is to monitor the busy status register of the Flash chip after setting the Write Enable Latch (WEL). If the WEL has failed to be set after 10 attempts (with 0.01 second delay between attempts), the program will exit, indicating the Flash chip was not successfully configured for writing. The high level operation of this program is shown below in Figure 16, which is an excerpt from the setup block of the program. This code is operating at the same time the shell script is sending the BMP data. This code first calculates the file size of the BMP based on the first four bytes received through the serial port. Then, a temporary buffer is used to store the incoming 256 byte packages and subsequently write these values to the flash chip in a loop that evaluates until the number of bytes received is no longer less than the size of the file calculated prior.

```
277  void setup() {
292    Serial.println("Send the BMP file...");
293    // Wait for the incoming data to start
294    while (!Serial.available()) {
295      delay(100);
296    }
297    // First 4 bytes are the file size in little-endian format
298    uint32_t fileSize = 0;
299    for (int i = 0; i < 4; i++) {
300      while (Serial.available() == 0) {}; // Wait for each byte
301      uint8_t byte = Serial.read();
302      String logMessage = "Byte " + String(i) + ": " + String(byte, HEX);
303      Serial.println("Byte" + String(i) + ": " + String(byte, HEX) );
304      //writeLog(logMessage);
305      fileSize |= ((uint32_t)byte) << (i * 8);
306    }
307    String fileSizeLog = "BMP file size received: " + String(fileSize) + " bytes";
308    Serial.print("Receiving BMP file of size: ");
309    Serial.println(fileSize);
310    uint8_t buffer[256]; // Temporary buffer for data chunks
311    uint32_t address = MUSIC; // Start address for writing to the flash
312    size_t bytesReceived = 0;
313    while (bytesReceived < fileSize) {
314      if (Serial.available()) {
315        size_t len = Serial.readBytes(buffer, min(sizeof(buffer), fileSize - bytesReceived));
316        writeFlash(address, buffer, len);
317        address += len;
318        bytesReceived += len;
319        Serial.print("bytesReceived: ");
320        Serial.println(bytesReceived);
321      }
322    }
323    Serial.println("BMP file transfer complete.");
324  }
```

Figure 16: High Level Operation of Flash Write Code

During the first attempt to prepare the chip, the WEL failed to set, indicating the Flash chip was either damaged or not communicating with the ESP32-WROOM-32E. After comparing the schematic to the layout and predetermined pin definitions, it was determined that the cause of the error was likely a mistake in layout. As mentioned above, the CS pin of the Flash chip needed to be routed to GPIO13 so as to not conflict with the CS pin on the LCD Driver (GPIO5). This change was never executed in layout, so although the .cpp program configured the Flash chip's CS pin to GPIO13, it was actually still connected to GPIO5, resulting in the ESP32-WROOM-32E being unable to communicate with the chip. To test this, the code was changed to redefine the CS pin to GPIO5. After doing so, the Flash chip enumerated properly and was successfully written using the shell script. At this point, the Flash chip was determined to be functional, but not fully functional as it would not be able to communicate in "parallel" with the LCD Driver as they shared the same CS pin. To resolve this issue, the trace connecting the Flash CS pin to GPIO5 was cut on the PCB and a wire was soldered to reroute the trace to GPIO13 on the ESP32-WROOM-32E. The program was reverted to its original pin definitions (GPIO13 for CS) and retested. The Flash chip was successfully programmed. To ensure full functionality, the image loaded to the flash chip was attempted to be displayed on the LCD Display. This was successful, indicating full functionality of the Flash chip.

3.6 Subsystem 3 (Speaker) Design and Operation

Requirements:

The Speaker Subsystem handles the I2S driven bluetooth protocol in order to play user-streamed music. This subsystem is fairly straightforward, as it only requires 1 GPIO pin, a speaker, and an audio amplifier with its external components. This bluetooth-enabled method of streaming music used only GPIO25, which is the Digital to Analog Converter (DAC) channel on the ESP32-WROOM-32E and interfaced with I2S. The speaker needs to have a small footprint with a low power consumption as this is not one of the main features of the Pixie. The audio amplifier is required to boost the output power so that the user is able to hear their music playing. This is because the internal DAC of the ESP32-WROOM-32E has a low output voltage of around 1.1V, so the amplifier was used to boost the output voltage level, which also corresponds to the output power level.. Pixie was designed with the idea of being a study aid, and the purpose of implementing the speaker was for a user to be able to play study music at a low level without causing distractions.

Major Components:

2030 8Ω Speaker:
The 2030 8Ω Speaker is used to play music generated by the user's phone or laptop through the ESP32-WROOM-32 E's ability to use Bluetooth streaming. This speaker has a relatively small blueprint (20x30x7mm) and is operable on Pixie's supply voltage of 5V. It acts like a bridge tied load.

Graphs:

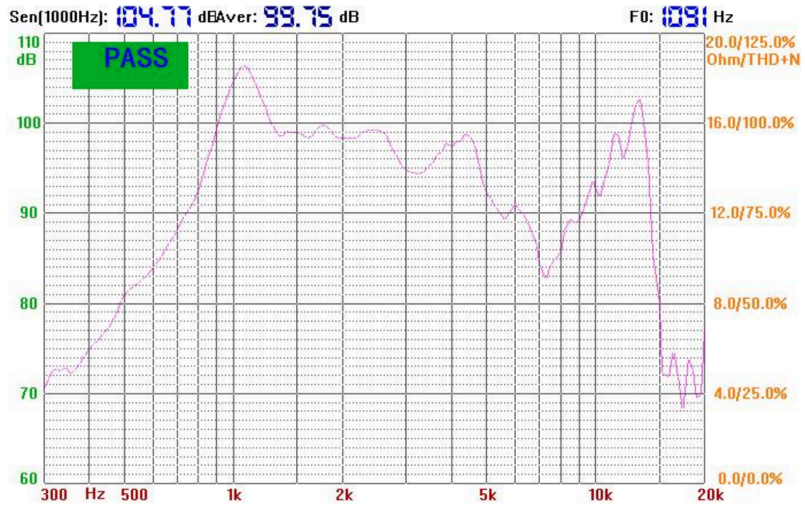


Figure 17: Frequency Response Graph of the 2030 Speaker

The speaker's frequency response graph shows that there is a low level of noise on the higher frequency end but not a large Sound Pressure Level (SPL) which is a perfect implementation for Pixie.

Audio Amplifier Module

The Audio Amplifier Module utilizes a HXJ8002 audio power amplifier chip to amplify signals with a low supply voltage and harmonic distortion. HXJ8002 amplifies audio signals without the use of any coupling capacitors or bootstrap capacitors. In addition, it has improved pop and clicks circuitry to significantly reduce transition noise when powering the module on and off. It is capable of delivering 1.5W to an 8Ω bridge tied load, which is the speaker in our application. With the bridge tied load, the HXJ8002 has one inverting output and one non-inverting output, which leads to double the voltage swing across the speaker and around 4 times the power for a given supply voltage. Because of this, the module consumes a low amount of power and is ideal in the Pixie. The module also uses a 10 kΩ resistor that forms an RC circuit with the parasitic capacitance of the ESP32-WROOM-32E to reduce noise with the internal DAC.

Schematics:

Reference Schematic:

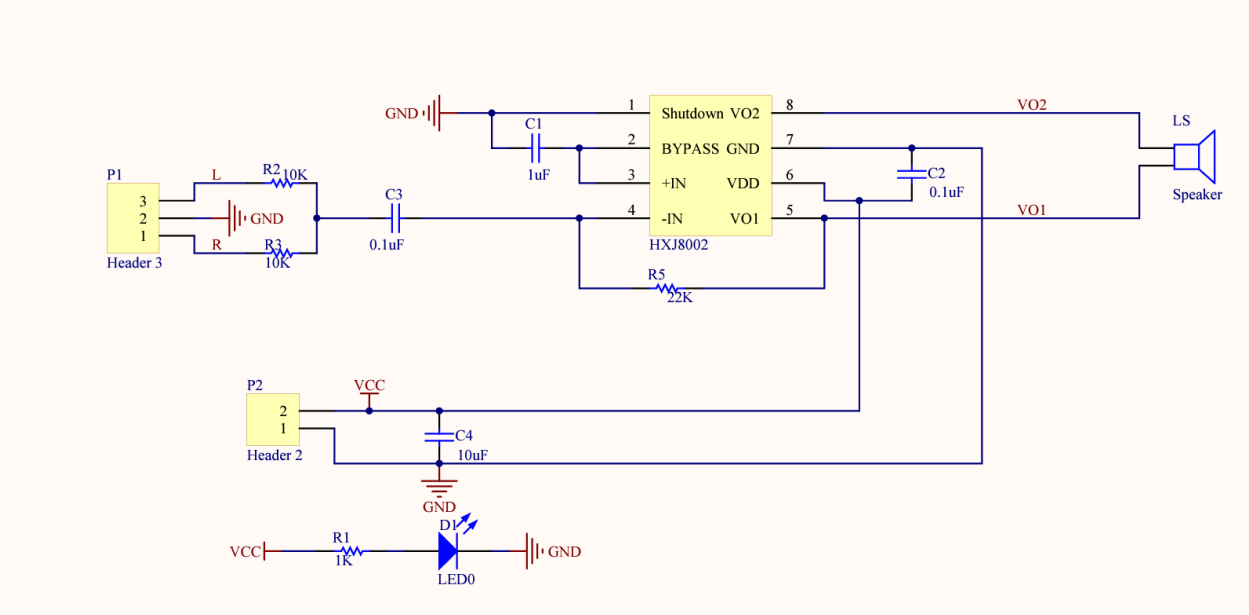


Figure 18: Reference Schematic of the Audio Amplifier Module

Figure 18 shows the reference schematic that was provided by the HXJ8002 datasheet. In this iteration of the schematic, it has a power LED, a 22kΩ resistor, and a 1x3 header pin for ground and the left and right channels of the output audio. It also has a 2 pin header in order to connect

ground and VCC to the IC. This schematic was used in fabrication of the breakout board that was used for the early prototyping stages of Pixie. In our final schematic, it was deemed not necessary to have both the left and right channels of the audio, so in Pixie's iteration we only have the left channel implemented and we did not incorporate the power LED onto our board since there was a main LED that would be powered on when the USB-C port was connected.

KiCad Schematic:

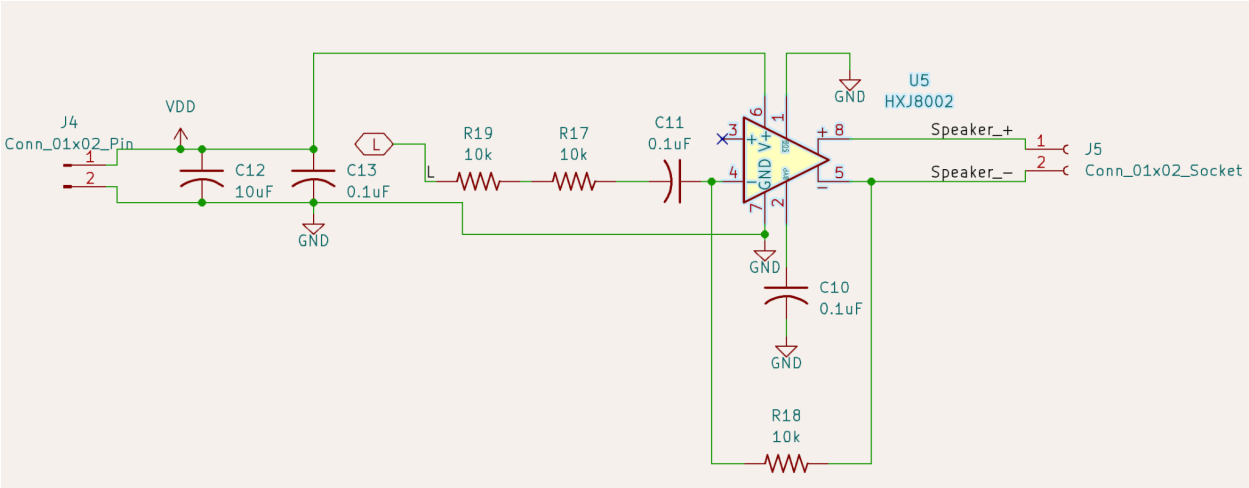


Figure 19: KiCad Schematic of the Audio Amplifier Module

In the KiCad schematic, there is an audio power amplifier that is used to represent the IC on the breakout board, since we could not find the footprint of the IC. It was ensured that the specs on the PCB layout were the same as the ones on the schematic.. Compared to the reference schematic, there are fewer header pins, since it is placed on the main board where the header pins are no longer of use. In addition to this, we did not use the 22kΩ resistor as we wanted the external components of to be simple so we placed 2 10kΩ resistors in series and there was no issues with the noise level that was demonstrated. The board did not need the 1x3 header pin anymore, since we were only routing the audio to the left channel and we did not want to waste space on our main board with header pins.

3.6.1 Subsystem 3 Testing

The speaker was first breadboarded with the audio amplifier module, 10kΩ resistor, and an ESP32-WROOM-32E. The audio amplifier module was on a breakout board with all of its required external components. With a simple sample code, the speaker was able to play music through a bluetooth connection to the ESP32-WROOM-32E. After the hardware setup was complete, the software needed to be refined in order to integrate it with the main code that ran all of the other Pixie tasks. The setup in order to play music via Bluetooth required the I2S to be

configured with the internal DAC of the ESP32-WROOM-32E. The code used a basic Advanced Audio Distribution Profile (A2DP) library that allows the ESP32-WROOM-32E to receive sound data from a bluetooth source. The audio is then played back through the speaker. AVRCP functions were then implemented in order to play and pause the music, and to increase and decrease the volume of the audio. There were some issues with the integration of the code into the main code, and this was mainly due to the fact that Bluetooth and WiFi could not run at the same time since the ESP32-WROOM-32E only has 1 antenna, so it can only handle 1 kind of data stream at a time, either Bluetooth or Wifi. There were core dumps encountered in the preliminary stages because of improper configuration of the I2S interface. It was initially thought that it was because it was running on the default core of the ESP32-WROOM-32E where the wifi tasks were running, which is core 1. Thus, we pinned it to core 0 and while that was a temporary fix, it still gave a core dump error. Once we found that it was due to incorrect configuration and with some debugging, the final software aspect of the speaker subsystem was complete and was able to stream music from a phone and a laptop.

3.7 Subsystem 4 (Keyboard) Design and Operation

Requirements:

The 6-key keyboard is responsible for managing and executing the user input for the Pixie. Each switch must be able to properly detect, respond, and control each of the Pixie functionalities based on the current active display mode. The debounce and logical control of the buttons are handled through the software, so it will not be described in detail in this section (code discussed in software subsystem).

Major Components:

Cherry MX Black Switches: MX1A-11NW:

The Cherry MX black switches are mechanical keyboard switches. There are only two pins on the switch, where polarity does not matter, and these switches operate by completing the electrical connection of a circuit when pressed down. These switches also have fixation pins to help with mounting the switches to the PCB. 10k pull-up resistors as well as an 8-pin JST connector are also implemented in the PCB.

Schematic:

The PCB housing the switches and their supporting components is distinct from the main PCB containing the microcontroller, as depicted in Fig. 20. Each board is outfitted with an 8-pin JST connector, facilitating connection from the keyboard PCB to the main board. Given the modest number of switches, one pin of each switch is directly routed to a GPIO pin (via the JST connector) instead of utilizing a diode matrix (Fig. 21). The remaining pin of each switch is grounded. To ensure functionality, a 10k Ohm pull-up resistor is installed on each GPIO pin. Consequently, when unpressed, the switch is pulled HIGH, and when pressed, the pin is grounded (LOW). The simplicity of the circuitry eliminates the need for diodes and decoupling capacitors; any required debouncing can be effectively managed through software debouncing. The switches are mapped to the GPIO pins seen in Fig. 22.

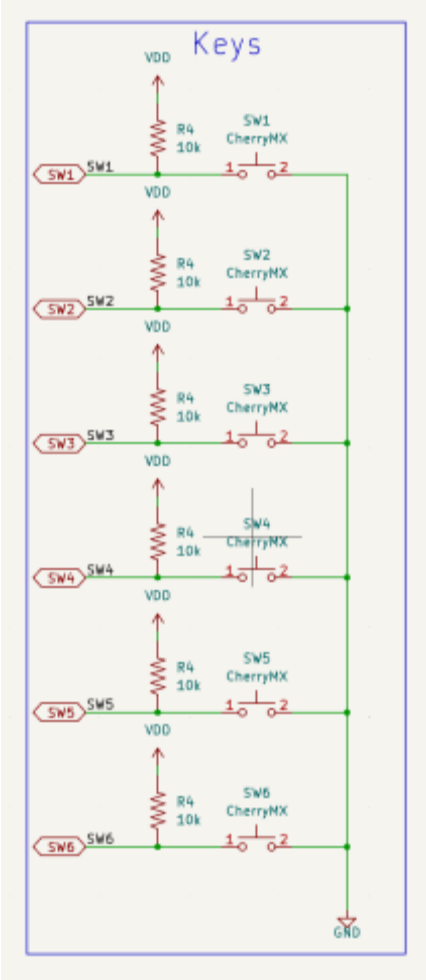


Figure 20: Schematic of switches wired with pull-up resistors

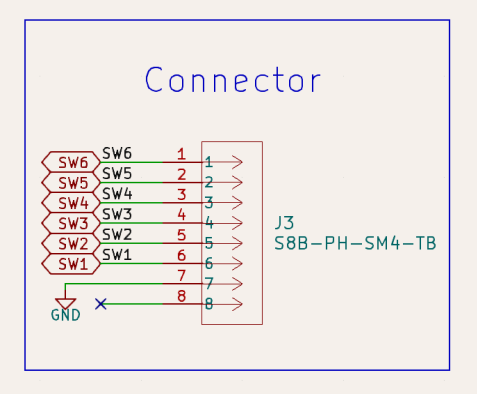


Figure 21: JST Connector Pin Assignment

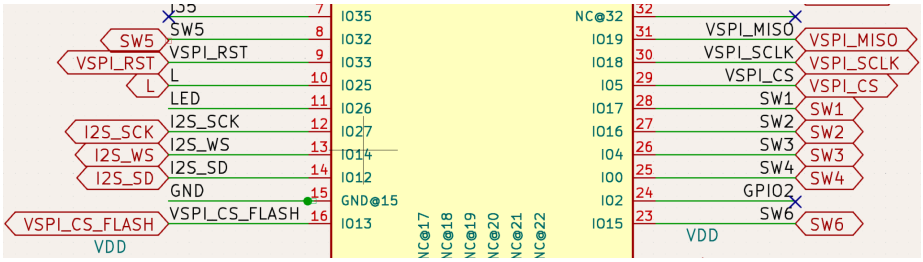


Figure 22: GPIO pin assignments on the ESP-32

3.7.1 Subsystem 4 Testing

Testing the functionality of the switches followed a straightforward process. Pushbuttons were connected according to the schematic on a prototype board. The code was crafted using Platform IO (Visual Studio Code), incorporating software debouncing. Initially, software debouncing utilized a temporary variable to track the button's last state. The millis() timer function measured the time elapsed between state changes. If this duration fell short of the specified debounce time (potentially due to noise), the timer reset. Conversely, if the duration exceeded the debounce time, the button press was acknowledged via an if conditional, executing the corresponding button logic. To validate button registration, LEDs were linked to arbitrary GPIO pins and programmed to toggle with button presses. This testing ensured the functionality of the debouncing code and confirmed compatibility with reading digital inputs on the designated GPIO pins. However, due to the inherent continuous nature of this code structure (utilizing a looped task for button polling), detection and debouncing code were subsequently refined through change notification interrupts and timer interrupts (discussed later in the software subsystem section).

3.8 Subsystem 5 (Wi-Fi/Website) Design and Operation

Requirements:

The Wi-Fi subsystem plays a pivotal role in the Pixie design, facilitating real-time display of weather conditions and user-generated notes based on the device's location. Its protocol must seamlessly connect to client servers to retrieve data from open APIs, ensuring accurate updates of current time, date, and weather. Additionally, Wi-Fi enables users to input and upload notes to Pixie's display via a server hosted by the device. To mitigate CPU workload, Wi-Fi integration necessitates careful management to activate the module only when required, optimizing efficiency and conserving resources.

Code Structure:

The built-in WiFi module on the ESP32-WROOM-32E utilizes the external antenna to operate at a frequency between 2412 to 2484 MHz. The WiFi communication module uses TX and RX pins on the chip to communicate with external devices on the board, in our case to communicate with external APIs and the display screen. (espressif.com)

Uploading Notes through Wi-Fi:

```
void handleUpload() {
  if (server.hasArg("text")) {
    String text = server.arg("text");
    Serial.println(text);

    // Clear screen
    tft.fillRoundRect(30, 30, 740, 430, 15, LIGHT_BROWN);

    // Display text on RA8875 TFT LCD
    tft.textMode();
    tft.setTextSize(1);
    tft.textEnlarge(1);
    tft.textSetCursor(40, 40);
    tft.textTransparent(RA8875_WHITE);
    tft.print(text);

    // Send response to client
    server.send(200, "text/html", "Text received: <b>" + text + "</b><br>Press button 2 to upload more notes.");
    received = true;
  } else {
    server.send(404, "text/plain", "Bad Request");
    received = true;
  }
}
```

Figure 23: Framework for Uploading User-Generated Notes

To enable users to upload notes, a web server transmits an HTML webpage to a web client featuring a textbox for text input. Users input their text, which is then uploaded from the website and displayed on the Pixie screen. Upon submission, the text is retrieved, displayed on the screen, and users can choose to upload more notes by pressing the button, returning them to the website's starting page. Following text display, the server sends a confirmation to the client, indicating successful receipt and display of the text. In Fig. 24, the code verifies the server for user input; if text is detected, it is displayed on the Pixie screen. A response is then sent to the client confirming text receipt, while any failure triggers a "Bad Request" message on the webpage, as depicted in Figure 15, which showcases the HTML code for the text-input website. Featuring a header labeled "PIXIE NOTES" and an expandable textbox with a submit button, this website facilitates user text input.

```
void handleRoot() {
  String htmlContent = "<!DOCTYPE html>\
                       <html>\
                       <head><title>PIXIE NOTES</title>\
                       <link href='https://fonts.googleapis.com/css2?family=Pixelify+Sans:wght@400&display=swap' rel='stylesheet'>\
                       <style>\
                         body { background-color: #CCAC; }\
                         h1 { font-family: 'Pixelify Sans', sans-serif; color: #72E8; font-size: 24px; text-align: center; }\
                         .form-container { display: flex; flex-direction: column; align-items: center; }\
                         textarea { color: black; background-color: white; width: 90%; height: 150px; padding: 8px; border: 2px solid #72E8; margin-bottom: 10px; }\
                         input[type='submit'] { \
                           font-family: 'Pixelify Sans', sans-serif; \
                           width: 20%; \
                           padding: 8px 15px; \
                           border: none; \
                           background-color: #72E8; \
                           color: white; \
                           cursor: pointer; \
                         }\
                       </style>\
                       </head>\
                       <body>\
                       <h1>PIXIE NOTES</h1>\
                       <div class='form-container'>\
                         <form action='/upload' method='post'>\
                           <textarea name='text'></textarea>\
                           <input type='submit' value='Submit'>\
                         </form>\
                       </div>\
                       </body>\
                       </html>";

  server.send(200, "text/html", htmlContent);
}
```

Figure 24: HTML Content Implemented for the Website

Requesting from the OpenWeather API:

To retrieve current weather information, local weather data is fetched from an open-source API. Figure 25 illustrates the function responsible for retrieving content from a specified server path, storing it as a character array (i.e., string). This function returns a string containing the retrieved contents. In the weatherAPI() function, as depicted in Figure 26, the returned string, if successful, is parsed and converted to a JSON file for easy data retrieval. Pixie requires only general weather conditions (e.g., Cloudy, Clear, Rain), current temperature, and day/night status. The snippet of code showcases extraction of correct information by specifying indexes from the JSON file. For instance, to extract the main weather condition, data is located within the

"weather" structure under "main." Other required data is similarly extracted. Since temperature is provided in Kelvin, it is converted to Fahrenheit before being displayed on the screen.

```
String httpGETRequest(const char* serverName) {
  WiFiClient client;
  HTTPClient http;

  // Your Domain name with URL path or IP address with path
  http.begin(client, serverName);

  // Send HTTP POST request
  int httpResponseCode = http.GET();

  String payload = "{}";

  if (httpResponseCode>0) {
    Serial.print("HTTP Response code: ");
    Serial.println(httpResponseCode);
    payload = http.getString();
  }
  else {
    Serial.print("Error code: ");
    Serial.println(httpResponseCode);
  }
  // Free resources
  http.end();

  return payload;
}
```

Figure 25: Function to pull JSON file from specified server path

```
JsonObject obj = doc.as<JsonObject>();


String weather = obj[String("weather")][0][String("main")];
String icon = obj[String("weather")][0][String("icon")];
float rawtemp = obj[String("main")][String("temp")];

updateWeatherCondition(weather, icon);
```

Figure 26: Part of the function - weatherAPI()) - to parse JSON file and extract required information.

3.8.1 Subsystem 5 Testing

Testing of this subsystem proceeded in sequential stages. Initially, emphasis was placed on ensuring proper Wi-Fi connectivity by verifying the Wi-Fi connection status (e.g., if WiFi.status() == WL_CONNECTED). Subsequently, attention shifted to confirming the setup of the client server, a straightforward process involving checking if the IP address for the client server loaded correctly. Development and testing of the httpGETRequest followed, with serial.println statements employed to assess proper transmission of content from the server. Once this step was validated, a function was devised to accurately convert character data into a JSON file, deserialize the content, and extract the desired data. This data was then utilized to update the variable containing current weather information.

3.9 Subsystem 6 (Software) Design and Operation

Requirements:

The primary software is tasked with providing directives and data for the Pixie to execute its core functions. It must adeptly configure, set up, and initialize all necessary data or functions for smooth operation. Moreover, the Pixie software must seamlessly execute each function outlined in the product description, such as playing music via Bluetooth, setting timers, and displaying notes, in response to user input via the keyboard switches. These functions should operate efficiently and reliably, necessitating careful program structuring to effectively manage and update data, monitor the program's current state (e.g., Bluetooth status, active display), and execute tasks promptly. This entails robust task management and error handling to enhance code reliability.

A fundamental requirement for the Pixie is the ability to cycle through display modes and execute other tasks based on button presses, contingent upon the active display mode. Switches 1 and 6 are designated for controlling the active display mode: switch 1 toggles the display mode forward until it wraps around to the first display mode, while switch 6 toggles the display mode in the opposite direction. Switches 2 through 5 are assigned distinct functions contingent upon the active display mode, delineated as follows:

	Display Mode 1	**Display Mode 2**	**Display Mode 3**	**Display Mode 4**	**Display Mode 5**
Switch 1	Toggle display	Toggle display	Toggle display	Toggle display	Toggle display
Switch 2	N/A	Upload notes	Toggle light	Toggle Bluetooth	Start/Stop timer
Switch 3	N/A	Upload pic 1	N/A	Play/Pause	Reset timer
Switch 4	N/A	Upload pic 2	N/A	Decrease volume	Decrease timer
Switch 5	N/A	Upload pic 3	N/A	Increase volume	Increase timer
Switch 6	Toggle display	Toggle display	Toggle display	Toggle display	Toggle display

Code Structure:

Smooth operation of the Pixie is contingent on the synchronization and organization of its tasks and functions Software structuring is paramount to ensuring a manageable CPU workload and the execution of functions as intended. The software serves as the linchpin, amalgamating each independent subsystem into a cohesive whole system. Given the complexity of the code, only the

key features will be elaborated upon: overarching task management, interrupt service routines,, and a brief overview of the RTOS tasks.

Task Management:

The most important feature of the Pixie software is to be able to manage and synchronize all of the separate functions the Pixie requires to operate smoothly. Tasks were created and managed through FreeRTOS, as the real-time operation is crucial for the functionality of the Pixie. To prevent unexpected behavior and chip rebooting, all tasks (as well as interrupt service routines) are event-driven and no one task is left to loop indefinitely without control. These events are monitored, managed, and controlled by flags as well as queues to hand off relevant data and status indicators. Each task is blocked from running unless given the control to execute (by receiving the instruction through a queue). For the majority of chip rebooting errors, it is due to the watchdog timer being triggered. The Watchdog timer is periodically reset in the Idle hook to ensure that no one task is starving the CPU of execution time; it is essential to have the Idle hook run periodically to perform "housekeeping" tasks, such as freeing memory, and ensure timely and efficient execution. The tasks are controlled in several different ways in order to satisfy the Watchdog timer.

First and foremost, it's crucial to ensure that no tasks are left running without supervision. Each task is designed to wait for an item from a queue using the xQueueReceive() function and portMAX_DELAY (see example function in Fig. 27). This function, equipped with the appropriate parameters, blocks a task until a new item in the queue is received. Upon receiving the item, the task is granted a single execution cycle before it is once again paused. This method of queue handling affords precise and efficient control over task execution, enabling each task to "take turns" executing.

```
if(xQueueReceive(buttonQueue, &pressed, portMAX_DELAY)){
```

Figure 27: Task waiting to receive new object in the "buttonQueue." portMAX_DELAY defines an indefinite waiting period

Additionally, the FreeRTOS tasks are regulated by task priority. Tasks are prioritized for execution, meaning that if two tasks are unblocked simultaneously, the one with the higher priority takes precedence and executes first before allowing lower priority tasks to proceed. Consequently, careful assignment of task priorities is essential to ensure critical tasks are executed promptly, while still allowing sufficient time for lower priority tasks to complete their operations. For the Pixie, the priorities of each task were selected based on the urgency of the tasks to execute. The Display Task is at the highest priority (4) in order to make sure that the display mode is updated in a timely manner and that all the data is properly initialized and

updated before any other tasks (that depend on the display mode) operate. The Button Handling Task is also configured with highest priority (4) to make sure that the Pixie is responsive to button presses. However, the button task is not at a higher priority than the display task to ensure that button presses do not interrupt any crucial initializations and data transfers that occur in the Display Task. The last task - the Bluetooth Task - is left at the lowest priority (1), as it is not a critical function. Furthermore, despite the lower priority, the task blocking via queues (as aforementioned) assures that the higher priority tasks will allow the low priority Bluetooth task to have CPU execution time in a reasonable time frame. See Fig. 28 for the three task configurations in the set up code.

The final measure to avoid triggering the Watchdog timer is to delegate each task to a specific core. Core 0 is much more sensitive to the WatchDog timer, so all of the tasks except for the Bluetooth task are run on Core 1. The Bluetooth Task was specifically created in order to pin this task to Core 0 so as to not interfere with other tasks running on Core 1 (see Fig. 28). Any additional tasks running on Core 0 often result in unwanted behavior, such as slow execution and SPI flash reboots.

```
// Create task for button handling
xTaskCreatePinnedToCore(
  ButtonTask,               // Task function
  "Button Task",            // Name of the task
  5000,                     // Stack size in words
  NULL,                     // Task parameters
  4,                        // Task priority
  NULL,                     // Task Handle
  1                         // Pinned to core 0 (CoreID)
);

// Create task for bluetooth
xTaskCreatePinnedToCore(handleBluetoothCommands, "handleBluetoothCommands", 2500, NULL, 1, &Task1, 0);

// Create task for updating display mode
xTaskCreatePinnedToCore(DisplayTask, "Display", 4000, NULL, 4, NULL, 1);
```

Figure 28: FreeRTOS tasks initialized and configured in the set up code; only the Bluetooth task is pinned to Core 0.

Interrupt Service Routines (ISRs):

In addition to the use of queues, ISRs are also to keep the code robust, reliable, and well-managed. Change notification interrupts as well as software timer interrupts were utilized to help synchronize tasks and interrupt tasks when needed. Rather than a continuous button-polling task (which would reserve an unnecessary amount of RAM as well as starve other tasks of CPU execution time), a change notification (CN) interrupt was attached to each GPIO pin that the buttons are assigned to (Fig. 29). Whenever there is a change in state on the GPIO pin, the

interrupt handler is called. Each switch has its own interrupt handler, where the button press is debounced and the switch number is sent through a queue to the Button Handler task for logic handling. The ISR priorities are higher than any of the other task and timer interrupt priorities to ensure that all other tasks are blocked and the operation is yielded to the ISR handler function. The function prototype for the change notification interrupt for switch 4 can be seen in Fig. 30.

```
// Attach interrupts
attachInterrupt(digitalPinToInterrupt(sw2), sw2_interrupt, FALLING);
attachInterrupt(digitalPinToInterrupt(sw3), sw3_interrupt, FALLING);
attachInterrupt(digitalPinToInterrupt(sw4), sw4_interrupt, FALLING);
attachInterrupt(digitalPinToInterrupt(sw5), sw5_interrupt, FALLING);
attachInterrupt(digitalPinToInterrupt(sw6), sw6_interrupt, FALLING);
```

Figure 29: Change Notification interrupts attached to each switch; triggered on the falling edge.

```
void IRAM_ATTR sw4_interrupt(){
  BaseType_t pxHigherPriorityTaskWoken = pdFALSE;
  unsigned long interrupt4_time = millis();
  if(interrupt4_time - lastDebounce4Time > 300){
    lastDebounce4Time = 0;
    buttons_pressed = sw4;
    xQueueSendFromISR(buttonQueue, &button_press, NULL);
    if (pxHigherPriorityTaskWoken) {
      portYIELD_FROM_ISR();
    }
  }
  lastDebounce4Time = interrupt4_time;
}
```

Figure 30: Change Notification interrupt handler function for switch 4.

The CN function prototype for switch 4 in Fig. 30 provides the general structure for each ISR handler attached to the switches. These CN interrupts are triggered when a falling edge (i.e., a button is pressed) is detected on the assigned GPIO pin. The button debouncing is done using the millis() function, checking to see if the state of the button (LOW) was maintained for at least the debounce time. This ensures that any unwanted noise or unintentional button presses do not affect the operation of the Pixie. Once the button press is appropriately debounced, the ISR records the switch that was pressed - in this case, switch 4 - and sends the variable to the buttonQueue for processing. Finally, the ISR is instructed to yield to the Button Handling Task that was awoken through the buttonQueue.

Timer interrupts are also configured in this code for functions that need to be continuously run, such as when the timer is active, or for updating the real time clock. Each timer interrupt is configured as a 1SHOT timer, where the timer only executes once, rather than executing

continuously after the timer period is reached. This ensures better control and management of when the timer interrupts are administered. The period of the timer is set to 1 second using the pdMS_TO_TICKS() function to update the current time (if the Home Display is active) and timer (if the timer is active) every second. See Fig. 31 for the detailed configuration parameters of the timer interrupts.

```
// Create timer interrupts
updateclock = xTimerCreate("Clock Update", pdMS_TO_TICKS(1000), pdFALSE, ( void * ) 0, updateClock);
updateTimer = xTimerCreate("Start Timer", pdMS_TO_TICKS(1000), pdFALSE, ( void * )0, updateTimerDisplay);
```

Figure 31: Software timer interrupt configuration in setup code to update the clock and the timer every second.

The timer is reset and enabled after the timer handler is called if there is no change in the display mode. This ensures that the Pixie's real time functionality continues to run without causing conflicts between other tasks and functions. Similar to the queues implemented for the tasks, these interrupts are carefully controlled by checking status indicators. Fig. 32 illustrates an example implementation of enabling the timer interrupt to update the current time. When the active display mode is on the Home Display (i.e., activeDisplayMode == 1), the timer interrupt to update the clock is reset and restarted. Thus, if the activeDisplayMode != 1, the timer is never re-enabled and remains deactivated until the Home Display is active again.

```
// Timer interrupt for clock update
void updateClock(TimerHandle_t xTimer){
  // Clear the time area (implement this as needed for your display)
  // For example, you might fill a rectangle with the background color over the time
  tft.fillRect(15, 10, 268, 65, BROWN);

  // Print the new time
  tft.textMode();
  tft.textTransparent(RA8875_WHITE);
  tft.textSetCursor(20, 10); // Reset cursor position if needed
  tft.setTextSize(1);
  tft.textEnlarge(3);

  if(checkTimeinfo()){…
  } else {…

  if(activeDisplayMode == 1){
    xTimerReset(updateclock, 0);
  }
}
```

Figure 32: Timer interrupt handler function prototype to periodically update the current time every second.

It is also important to keep track of the active tasks, functions, and interrupts to make sure that incompatible tasks are not attempting to execute at the same time. For example, bluetooth is unable to run when wifi is running. In order to prevent this conflict, the status of Bluetooth and Wi-Fi is always checked with a boolean variable acting as a status flag (e.g., bluetoothActive = true) before activating either function so that there are no conflicts. This is a form of error handling that is important to make sure errors are properly handled for any unexpected behavior and correct it.

Tasks:

Display Task

The Display Task is responsible for updating and initializing the display mode in response to specific button presses. Switch 1 and switch 6 are responsible for toggling between the display modes. The ISR handlers send an item to the displayQueue when a button press from switch 1 or 6 is detected. This unblocks the Display Task and accordingly updates the LCD screen to the new display mode (using the updateDefaultDisplay() function). The task uses a switch case statement to identify the current active display mode.

```
// Update Display Mode Task (sw1)
void DisplayTask(void *parameter) {
  int displayMode;
  while (true) {
    if(xQueueReceive(displayQueue, &displayMode, portMAX_DELAY) == pdPASS){…
    vTaskDelay(100/ portTICK_RATE_MS);
  }
}
```

Figure 33: Display task function prototype; execution of the task is dependent on receiving an item on the displayQueue.

Furthermore, the Display Task is also responsible for checking any status flags as well as blocking any other tasks, interrupts, or functions when the display mode is updated. This ensures that nothing can disrupt or interfere with critical data initialization or updating during a display mode change.

Button Handling Task

The Button Handling Task is responsible for processing the button press logic. Switches 2 through 5 send an item to the buttonQueue when a button press is detected. This triggers the Button Task to execute. Again, a switch case statement is used to identify which button was pressed and accordingly execute the corresponding function. Custom functions (e.g., button2(activeDisplayMode), button3(activeDisplayMode), etc.) are called to check the current

display mode since switches 2 through 5 execute different functions depending on the active display mode. For instance, switch 2 activates or deactivates the Bluetooth module when on Display Mode 4 (Music Display), while the same switch starts or stops the timer when on Display Mode 5 (Timer Display). More details for each of these custom button functions can be viewed in the source code.

```
// Main Button Handler Task
void ButtonTask(void *parameter) {
  int pressed;
  int button;
  // Loop forever
  while (true) {
    if(xQueueReceive(buttonQueue, &pressed, portMAX_DELAY)){
      button = buttons_pressed;
      switch(button){
        case sw2:
          button2(activeDisplayMode);
          break;
        case sw3:
          button3(activeDisplayMode);
          break;
        case sw4:
          button4(activeDisplayMode);
          break;
        case sw5:
          button5(activeDisplayMode);
          break;
        default:
          break;
      }
    }
    vTaskDelay(50/ portTICK_RATE_MS);
  }
}
```

Figure 34: Button Handling Task function prototype; execution of the task dependent on receiving an item in the buttonQueue (sent from any of the switch CN interrupt handlers).

Bluetooth Task:

The third and final RTOS task is the Bluetooth Task. Unlike the critical nature of the Display Task and the Button Handling Task, the Bluetooth Task is non critical and succinct. Like all of the other tasks, the Bluetooth Task is blocked until an item is received in the bluetoothQueue. This item is sent when switch 2 is pressed in Display Mode 4 (Music Display). The task also checks and records the status of the Bluetooth through a boolean variable (bluetoothActive).

```
///************************ BLUETOOTH ******************************///
void handleBluetoothCommands(void* Parameter) {
  while (true) {
    if (xQueueReceive(bluetoothQueue, &bluetoothActive, portMAX_DELAY) == pdTRUE) {
      if(bluetoothActive){
        a2dp_sink.end();
        Serial.println("Bluetooth deactived");
        bluetoothActive = false;
        bluetoothPlay = false;
      } else if (!bluetoothActive){
        a2dp_sink.set_i2s_config(i2s_config2);
        a2dp_sink.set_i2s_port(I2S_NUM_0);
        a2dp_sink.set_pin_config(pin_config2);
        a2dp_sink.start("ESP32_Bluetooth");
        Serial.println("Bluetooth Actived");
        bluetoothActive = true;
      }
    }
    vTaskDelay(100 / portTICK_PERIOD_MS);
  }
}
```

Figure 35: Bluetooth Task function prototype; execution of task is dependent on receiving an item in the bluetooth Queue.

3.9.1 Subsystem 6 Testing

Testing of the primary software commenced in the latter part of the project timeline, contingent upon thorough understanding and independent functionality of all subsystems. Each subsystem was systematically integrated to detect and resolve any issues. Initially, five display modes were designed, with a single button designated to toggle through them. The home display prioritized showcasing current time, date, and weather, necessitating integration of the WiFi subsystem to fetch data from an open source API. Task implementation focused on ensuring continuous display updates every second. The upload notes mode required parsing text from the client server and displaying it on the screen. The night light mode simply toggled a boolean. Integrating the music display mode involved Bluetooth module integration alongside peripherals like speakers and audio amplifiers, with each function mapped to specific switches. The final mode, though not requiring new subsystem integration, posed a more intricate, nuanced challenge, with complexities difficult to fully document. This iterative approach allowed for incremental addition and understanding of each feature, facilitating resolution of unexpected problems and refining the code for harmonious, error-free operation.

3.10 User Interface and Graphical Design

The user interface and graphics design of the Pixie play a vital role in shaping the branding and identity of our product. Aspiring to be a charming and approachable study aid and desk accessory, the aesthetic appeal and user interface serve as fundamental elements in achieving Pixie's overarching goal.

Aesthetic Inspiration:

The aesthetic vision for Pixie draws inspiration from miniature replicas of early 2000s desktop computers, evoking a sense of nostalgia reminiscent of 8-bit games. In line with this theme, all graphics are meticulously crafted to emulate pixelated digital art. Additionally, to ensure Pixie's visual appeal is gentle on the eyes, a soft color palette featuring shades of browns, whites, and greens was curated, lending a natural and calming ambiance. Leveraging the capabilities of the LCD screen, which supports RGB565, each color was carefully selected and converted into its corresponding hexadecimal value (Fig. XX).

```
#define LIGHT_BROWN 0xCCAC
#define LIGHTER_BROWN 0xEDD0
#define DARKER_BROWN 0x72E8
#define LIGHT_GREEN 0x8E25
#define LIGHT_ORANGE 0xFDEC
#define YELLOW 0xFF0E
```

Figure XX: Hexadecimal values for the color palette used for the Pixie Display

Display Modes:

Figure XX: Home Display

Figure XX: Music Display

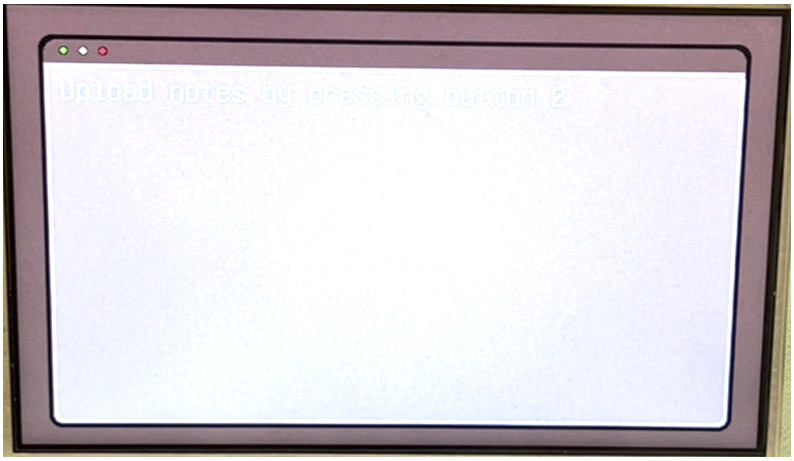


Figure XX: Notes Display

Every display mode was conceived with a minimalist approach, aiming to present essential information in a concise manner. A glimpse of a couple of these modes is provided, spanning from Fig. XX to Fig. XX.

Button Symbols:

In keeping with the theme of our aesthetic inspiration, each button was fashioned after 8-bit graphic buttons. These on-screen buttons not only serve to identify the functions they represent but also offer a visual cue of the associated function's status. Leveraging the basic drawing and shape-filling functions available for the LCD display, we meticulously crafted these buttons. Fig. XX showcases an example of the programmed graphics, where different shapes and lines are strategically placed and overlapped to create various icons, including the buttons. The buttons themselves are constructed by layering several shapes. Initially, a black rounded rectangle is

drawn, followed by a slightly smaller rounded rectangle, creating the illusion of a black outline. Subsequent color additions complete the pixelated button appearance.

Figure XX: Button design (not currently pressed)

Figure XX: Button design when pressed (flashed for a couple hundred milliseconds)

Moreover, each button has a corresponding "pressed" version, redrawn to simulate button depression and signal to the user that their press has been registered. Additionally, the status indicators, such as Bluetooth activation, are dynamically updated on the buttons, providing users with real-time feedback. The button graphics are shown as an example in Fig. XX and Fig. XX.

Icons:

Additional graphical designs were crafted following a similar methodology as the buttons. For instance, Fig. XX displays a custom weather icon depicting cloudy weather, alongside an icon representing a computer window screen. Similar techniques were employed to create various other weather icons, tailored to different weather conditions obtained from the open weather API. Fig. XX showcases a sample code snippet illustrating a function responsible for drawing a sun. This process of designing graphical elements ensures visual coherence and clarity throughout Pixie's interface, enhancing user experience and engagement.

```
void iconSun(Adafruit_RA8875& tft, uint16_t x, uint16_t y, uint16_t r)
{
  tft.drawLine(x-r*1.75, y, x+r*1.75,y, LIGHT_ORANGE);
  tft.drawLine(x, y-r*1.75, x, y+r*1.75, LIGHT_ORANGE);
  tft.drawLine(x - r*1.25, y - r*1.25, x + r*1.25, y +r*1.25, LIGHT_ORANGE);
  tft.drawLine(x - r*1.25, y + r*1.25, x + r*1.25, y - r*1.25, LIGHT_ORANGE);
  tft.fillCircle(x, y, r*1.2, LIGHT_BROWN);
  tft.fillCircle(x, y, r*1.1, RA8875_WHITE);
  tft.fillCircle(x, y, r, LIGHT_ORANGE);
  float offset = 0.9;
  tft.fillCircle(x, y, r*offset, YELLOW);
}
```

Figure XX: Example custom function to draw a sun icon on the LCD

Figure XX: Weather and computer window icon design

Full Resolution Images:

In addition to the natively drawn graphics and animations, and mentioned several times prior in this report, Pixie is also capable of displaying full resolution images stored in the Flash memory chip. Figure XX and XX below depict the function used to draw images to the LCD screen from the Flash memory chip. This function takes arguments for the address in the Flash memory chip where the image is stored, and the coordinates on the LCD Display to begin drawing the image. The function draws the image row by row using hardware accelerated functions provided by the LCD Driver. Figure XX shows the first half of the function, which uses the readFlashData() function (not shown, but similar in structure to the program used to write data to the Flash in the Subsystem 2) to parse relevant information in the BMP header. The program expects images to be 24-bit in depth, but converts them to 16-bit RGB565 format for display to the LCD. Each 24-bit color pixel is represented by 3 adjacent bytes in the Flash memory chip. Thus, each row of the BMP image is represented by 3 times as many bytes as there are pixels in the row. A buffer of this size (rawRow) is used to read in this many bytes from the Flash memory chip. Another buffer the same size as the image width is prepared to store the converted RGB565 color data. Figure XX shows the second half of the function. Here, the function uses the pre-allocated arrays to store the raw image data from the Flash and the RGB565 converted data to be drawn to the

screen. A loop iterates across the height of the image. In each iteration, the corresponding row of the image is read in, converted, and drawn to the LCD Display. The address of the row to be read in is incremented the appropriate value each iteration of the loop. After drawing the image, the dynamically allocated memory arrays are freed.

```
159 void drawBMPFromFlash(uint32_t bmpAddress, uint16_t x_start, uint16_t y_start) {
160     // Buffer for reading the BMP header
161     uint8_t bmpHeader[54]; // Standard BMP header size
162     readFlashData(bmpAddress, bmpHeader, sizeof(bmpHeader));
163     // Extract relevant fields from the header
164     uint32_t pixelDataOffset = *(uint32_t *)&bmpHeader[10];
165     int32_t width = *(int32_t *)&bmpHeader[18];
166     int32_t height = *(int32_t *)&bmpHeader[22];
167     uint16_t bitDepth = *(uint16_t *)&bmpHeader[28];
168     Serial.println(bitDepth);
169     if (bitDepth != 24) {
170         Serial.println("BMP is not 24-bit depth");
171         return;
172     }
173     // Calculate the size of one row of raw pixel data
174     uint32_t rowSize = (width * 3 + 3) & ~3;
175     uint8_t *rawRow = new uint8_t[rowSize]; // Allocate memory for one row
176     // Allocate memory for the RGB565 pixel array for one row
177     uint16_t *pixelRow = new uint16_t[width];
178     // Determine if the BMP image is stored top-down or bottom-up
179     bool topDown = height < 0;
180     height = abs(height);
181     // Adjust starting Y coordinate based on image orientation
182     uint16_t y_offset = topDown ? y_start : y_start + height - 1;
```

Figure XX: First Half of BMP Draw Function

```
183     for (int y = 0; y < height; y++) {
184         // Read one row of raw pixel data from the flash
185         uint32_t rowAddress = bmpAddress + pixelDataOffset + (topDown ? y * rowSize : (height - 1 - y) * rowSize);
186         readFlashData(rowAddress, rawRow, rowSize);
187 
188         // Convert to RGB565 and store in pixelRow
189         for (int x = 0; x < width; x++) {
190             int idx = x * 3;
191             uint8_t blue = rawRow[idx];
192             uint8_t green = rawRow[idx + 1];
193             uint8_t red = rawRow[idx + 2];
194             pixelRow[x] = rgb888ToRgb565(red, green, blue);
195         }
196         // Draw the row on the screen
197         tft.drawPixels(pixelRow, width, x_start, topDown ? y_offset + y : y_offset - y);
198     }
199     // Clean up
200     delete[] rawRow;
201     delete[] pixelRow;
202 }
```

Figure XX: Second Half of BMP Draw Function

4 System Integration and Testing

4.1 Testing integrated subsystems

After each subsystem was tested separately from the other subsystems, we slowly began integrating all components. First, the Display subsystem was integrated with the microcontroller and the Keyboard switches. The switches were assigned to different functions within the code to iterate through the controls of the display depending on the screen being shown. This initial subsystem integration testing is where we learned the most important part of our product design, task management.

Next, we integrated the keypad WiFi and Website subsystem using the on board WiFi modules. Again, most of the troubleshooting and integration changes came from organizing the tasks in our software to update the time and weather every thirty minutes or whenever the home screen was called back on the display screen. We learned that the on board WiFi module draws a lot of power and cannot be run while bluetooth is running. Therefore, the WiFi module is called when the update weather/time function is called, or when the Pixie is on the notes screen section where the user uploads notes via WiFi from the website.

Then, the speaker subsystem was integrated. This feature connected with the external bluetooth module and amplifier on our main board. Bluetooth is connected to the display on the bluetooth screen. Once integrated with the code, testing took place by connecting phones and personal computers to the Pixie's bluetooth to play music. Both phones and PCs and any music playing streaming service work on Pixie.

The Microphone subsystem was the most challenging part of the integration process. The Microphone software and subsystem works well on its own, but when we integrated the software with the rest of the code, the clapping to activate the night light feature simply did not work. We demonstrated the microphone subsystem working on our prototype, breakout boards. We also have the night light feature on the Pixie, but it is accessed by switching the light on and off with a key cap. The microphone is installed on the Pixie device, so when future enhancements fix the software issue with the microphone code integration, a simple software update will allow the subsystem to operate efficiently.

Finally, the Software subsystem and the Flash subsystem were integrated and tested with the entire Pixie interface during the entire subsystem integration process. Software was continuously being developed with each additional subsystem. The Flash was also tested coherently with the software.

4.2 Testing Satisfying Design Requirements

Upon full integration, testing, and demonstrations, it is evident that Pixie meets the Design Requirements written in Section 2 of this document. Pixie satisfies all hardware features and displays full software integration between subsystems. While the only discrepancy is the microphone feature working fully with the rest of the code, the purpose of the microphone (turning on a light) is still completed with the current design of the product. Each subsystem satisfies its own requirements. Therefore, the requirements for the Pixie as a whole are satisfied.

5 User Manual and Installation

1. Power the Pixie by plugging a USB-C cord into the USB-C power terminal on the side of the device. The Pixie's home screen will be displayed.
2. Sync the Pixie to your WiFi. Hold the left, top key down to sink the WiFi
3. The home screen display the date, time, weather based on your WiFi location. Additionally, a factory installed image will be displayed on the right of the screen.
    a. At any time you wish to update the weather, press the top right keycap. (Key 3)
4. The bottom, right key cap (switch 6) switches between screens displayed on the Pixie's face. The screens are displayed in the following order:
    a. Home Screen
    b. Bluetooth Screen
    c. Study Time Screen
    d. Notes (Website) Screen
    e. Night Light Screen
5. The Bluetooth screen shows another fun, factory installed image and 6 corresponding button features. Press the middle top button (switch 2) to turn on bluetooth. Look for the ESP32 Bluetooth connection on your personal device to connect to the Pixe.
    a. Press the bottom left two keys (switches 4 and 5) to control speaker volume.
    b. Press the right, top key to play/pause music.
    c. Bottom right switch takes you to the next screen.
6. The study timer screen shows a countdown clock and the following functions to control the timer.
    a. The bottom, left two keys increase, decrease the time displayed on the screen.
    b. The middle top key starts the timer
    c. The right top key stops the timer
    d. Bottom right key takes you to the next screen
7. The notes screen displays any notes uploaded by the user via the Pixie Website.

    a. Go to the Pixie website on your personal computer. In the white text box that is displayed, type your messages, important ideas, to do lists, etc. Then, press the upload button on the website. Your text will automatically be displayed on Pixie!
8. The Night Light screen is a dark, black screen. This can be considered Pixie's sleep mode.
    a. To activate the screen;press the center, top keycap and the screen will light up your room!
9. To completely turn off the Pixie, simply unplug the power cable. The Pixie will recall your WiFi Connection the next time you plug it in under the same WiFi address.
10. Happy Studying!

6 To-Market Design Changes

Our final version of Pixie provides users with the core features determined at the outset of the project. However, improvements to these features and additional features would make Pixie a more useful and attractive alternative to a cell phone as a study tool.

The first change in a marketable version of Pixie addresses improvements in the existing night light feature. We would like to fully implement the sound-activation of the nightlight, which was omitted in our final deliverable as it presented challenges in successfully executing the integrated software. This feature will enable Pixie to solve the issue of users instinctively reaching for their phone as a nightlight (and subsequently becoming distracted) by making the activation of its nightlight completely hands-free.

The next change is to include options for the user to page through images stored in the flash memory chip. Currently, users are only able to display a single image of their on each applicable display mode. However, users may desire the option to dynamically change the image on the same display mode. This change is fairly straightforward and only requires the configuration of a keypad switch to access different memory locations in the flash chip to redraw images.

During our presentation, several evaluators asked about emergency weather condition alerts in the case of incoming natural disasters, such as a tornado. We would like to implement this feature into Pixie so that even when users aren't in the vicinity of their phone, they still receive the safety benefits of emergency warnings.

While we already have a notes forwarding method for users to record important reminders/tasks while away from their desk, we would like to implement an additional feature to forward critical notifications from the user's phone such as emails and text notifications from important people. This change entails a significant amount of software development to link user's text messages/emails and filter through "important" senders. However, this would make abandoning one's phone in favor of Pixie as a study tool a much more attractive option for college students.

7 Conclusion

The development of Pixie is representative of a momentous accomplishment in designing a “study buddy” that enhances a student’s productivity. The idea of Pixie was conceived as a response to the distractions that are on a student smartphone. Pixie provides a multitude of tools without the distraction of social media. It includes a timer, bluetooth music streaming, night light, and a real date/time/weather display which are features that are conducive for school work and/or studying.

Throughout this project, our team encountered a range of challenges when integrating the individual subsystems. This required a thorough understanding of the hardware and software interactions and it was demanding for our team, given our background in electrical engineering without a lot of software experience. We overcame these challenges and all but one of our main features were able to be included in the final product.

The most challenging aspect of this project was integrating the sound-activated night light. It worked independently, but when incorporating it into the main code, software issues arose when it interacted with the bluetooth functionality. In addition to this challenge, ensuring that Bluetooth and WiFi operated seamlessly, given the ESP32-WROOM-32E’s single antenna, required attentiveness to task management and prioritization to avoid core dumps.

Our feelings about Pixie are positive. As a team, we are proud of the overlap of creativity and technicality that we demonstrated in Pixie’s development and are pleased with its functionality. It serves as a practical tool and a testament to the hard work that our team accomplished. Pixie’s design makes it an attractive addition to a student work area.

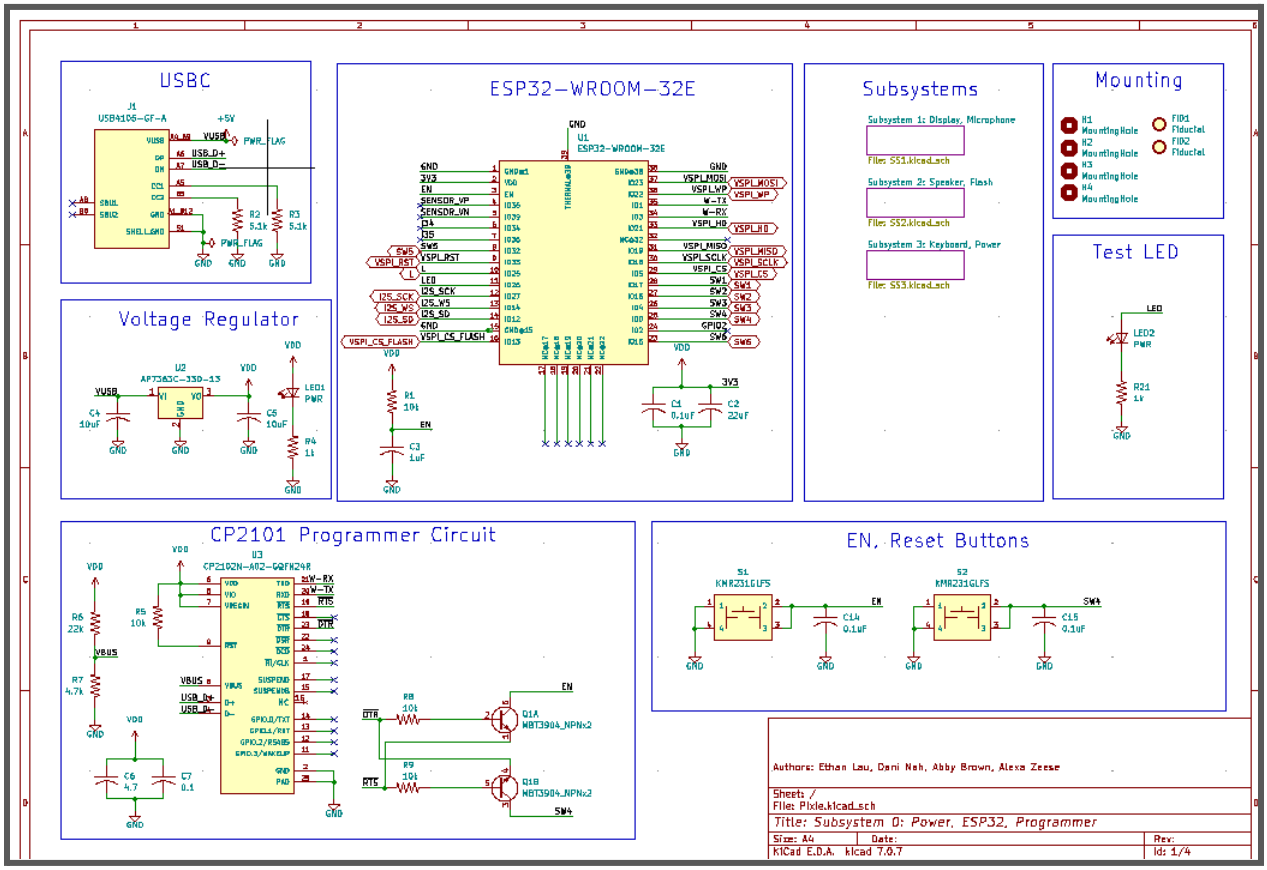
Pixie was an educational and challenging project. It provided out team with valuable insights into product design and integrated many aspects of engineering. The experience prepared us for future projects that require the same level of creation and interdisciplinary collaboration, which reinstates the foundations of engineering of problem-solving and development.

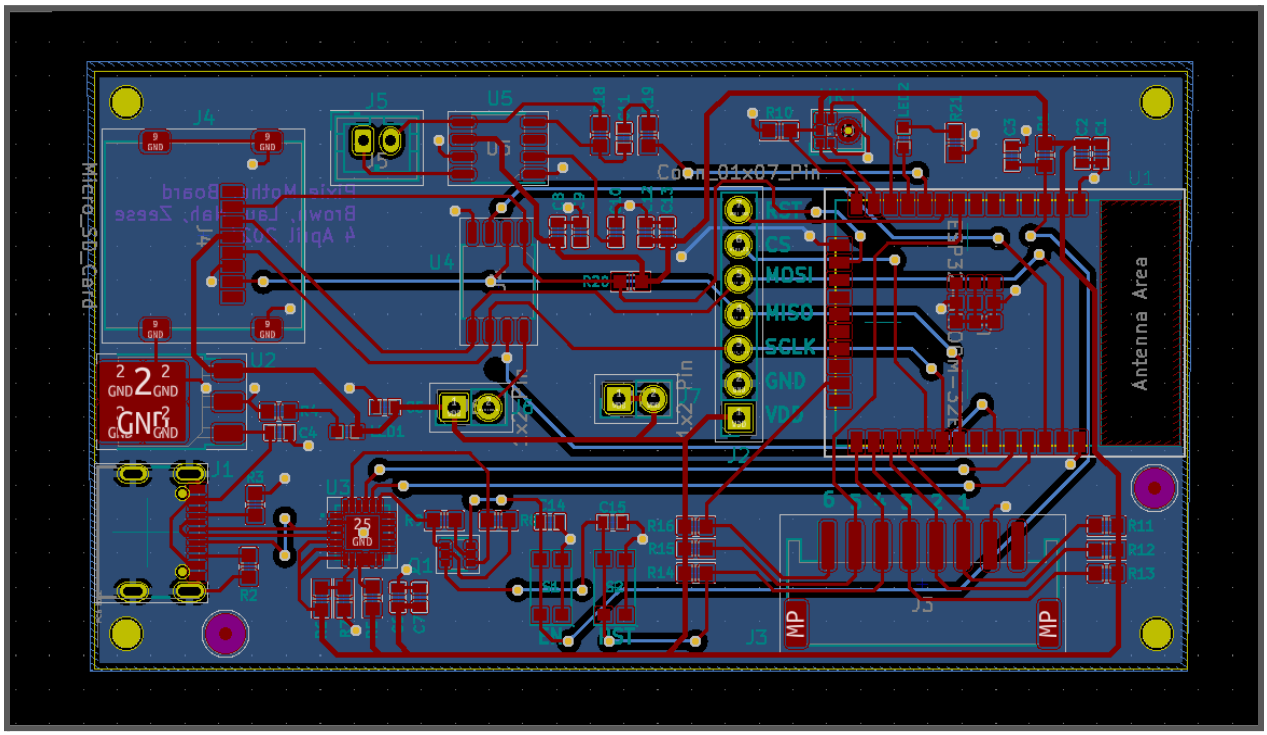
In conclusion, Pixie demonstrates how technology can be fit to address certain user needs while mitigating potential distractions. It stands as a promising model for future iterations in productivity devices.

8 Appendix

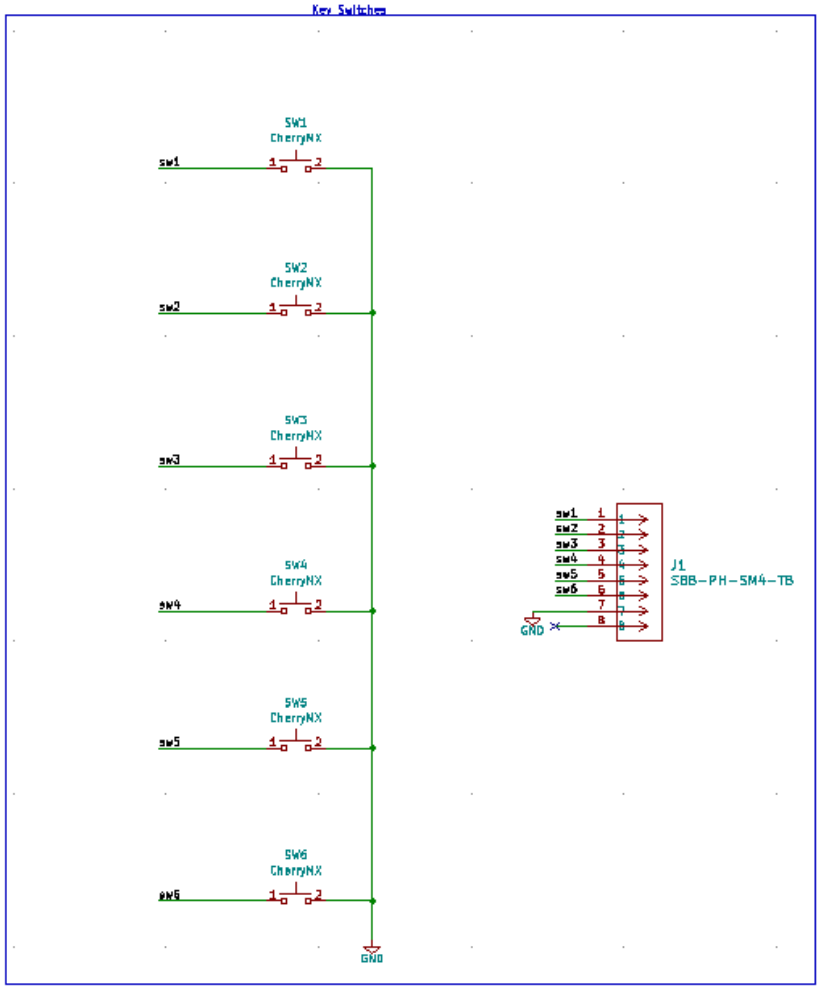
8.1 Complete Hardware Schematics

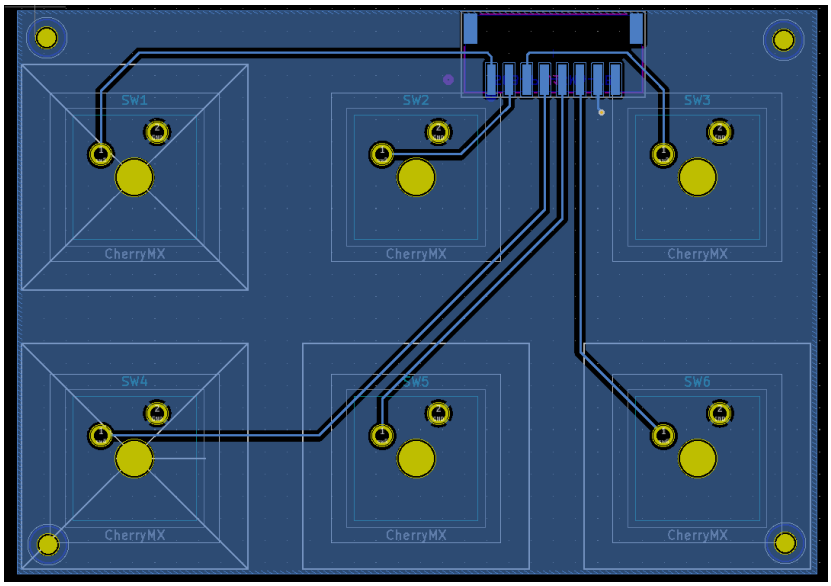
Main Board:



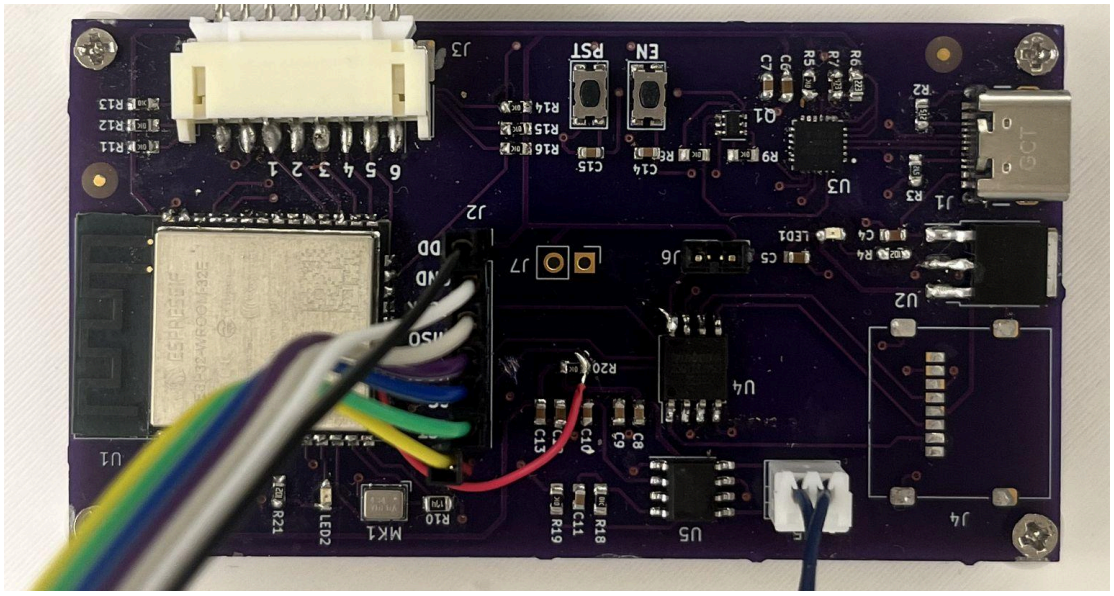


Keyboard SChematic & PCB:





Main Board and Keypad Built:



8.2 Complete Software Listings

The Code is rather large, you can access the entire listing here.
https://seniordesign.ee.nd.edu/2024/DesignTeams/pixie/files/PixieSRCfiles.zip

8.3 Work Cited/Data Sheets

1. ESP32-WROOM-E data sheet
   https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf
2. Bill Of Materials with links to additional data sheets:
   https://docs.google.com/spreadsheets/d/1nOjsUcRr5YgX-wwoyY2KqtLnSfijROXuSRDt720DhBE/edit?usp=sharing