Multi-Adjustable Port Station (M.A.P.S.)

Hunter Volonino, Sebastian Sotomayor, Jonathan Luna, Siya Sharma

Dept. of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 32816

***Abstract* — This project addresses a gap in charging technology, where device battery health may not be prioritized during situations when a (possibly older) device is left plugged-in for long periods of time. The main focus of this project is electronics hobbyists, but the working principles can be applied to many other use cases. For our case, we are demonstrating a smart four-port USB charger that can preserve battery health by turning off the charger once the charging current drops below a set threshold. This is intended to work with a wide variety of devices that charge over USB. We will accomplish this by integrating together current measurement, current switching, power delivery, and computing hardware.**

Index Terms* — *Lithium-ion battery charging, mobile devices, smart chargers, real-time systems, embedded systems, power delivery and regulation.

I. Introduction

Device hobbyists that collect various types of electronics exist to use and enjoy different portable electronics from different points in time. In particular, any device hobbyist with a large amount of mobile or portable devices must keep them charged if they desire to use them. Most portable electronic devices contain rechargeable lithium-ion batteries, which carry a set of risks in use. However, keeping a device plugged in all of the time while it is unused can degrade the battery and even cause it to expand. Charging a battery to 100 percent of its capacity also can eventually cause accelerated wear over time, and keeping a battery in a low state of charge is perhaps even more detrimental [1].

Our senior design project is designed for such hobbyists with various different devices that they use on a semi-regular basis. A device may be desired every few days, or every few weeks. To use the devices, they must be charged to a usable capacity, but not kept charging to the point where excessive battery wear happens.

Another important consideration is that, as this project is designed for various different types of devices, the battery's capacity cannot be read in a direct and standardized manner over USB or any other charging medium. Thus, the only things that can be directly read are voltage, current, and power draw while charging. It is up to the user to configure the charger based on that data, so the charger can be optimized to the user's preferences.

Our solution to this problem consists of turning off the charger at a set point. As a typical mobile device charges, the charging current draw drops. This is shown by Figure 1 below, with a sample charging session of an iPhone 6s attached to a USB multimeter. A user can set the minimum current threshold at which the device stops charging. We can measure the charging current, and allow the user to determine this threshold based on charging data. Once that threshold is determined and set, our charger can turn off the device's charging port for a user-set time, before turning on the port again and once again reading the charging current. This allows a configurable charger that can work with many different types of devices.

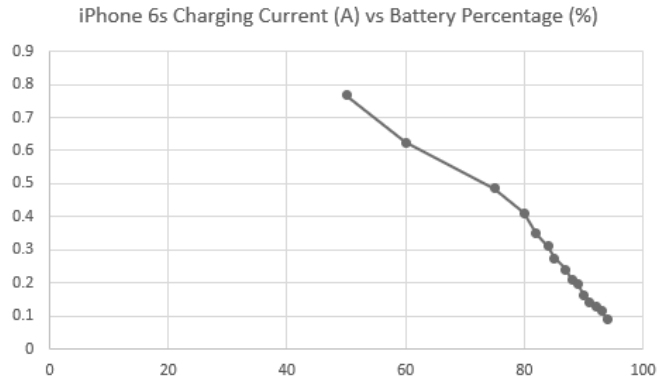

Fig. 1. Sample charging session of an iPhone 6s from 50% to 100%. The horizontal axis shows the device's indicated battery percentage, and the vertical axis shows charging current as obtained by a USB multimeter. This graph illustrates that as a device charges, the charging current drops visibly and detectably.

II. System Components

A. ESP32 Microcontroller

We have selected the use of an ESP32 microcontroller for this project. The ESP32 will control our current switching hardware, acquire data from our current measurement hardware, and send/receive data from the single-board computer hosting our user interface. Initially, we considered the use of FPGAs, but the ability to use a microcontroller that is much cheaper and simpler to program outweighed it. Most microcontrollers are RISC as it affords better power efficiency, which is also imperative for us.

The ESP32 is a family of microcontrollers created by Espressif Systems with a 32-bit architecture. Various different models exist with either Xtensa or RISC-V processors. RISC-V processors are only available in single-core, but Xtensa processors are available with two cores. All ESP32 microcontrollers have wireless capabilities via Wi-Fi and/or Bluetooth. Some ESP32 microcontrollers have the antennas integrated into the PCB, and others have a connector to add an external antenna. The ESP32 is available both as a bare microcontroller, or as a development board. Various product families exist, including WROOM, SOLO, MINI, and WROVER. For this project, we will be using the ESP32-WROOM-32E. It has a dual-core Xtensa LX6 processor, paired with 520 KB of RAM, and the option for 4, 8, or 16MB of flash storage. It is equipped with 802.11n Wi-Fi capability and Bluetooth 4.2, and has 26 GPIO pins. It has PWM support and an ADC, along with UART, I2C, and SPI capability. As for programming with an integrated development environment (IDE), it can be programmed with the Arduino IDE, or Espressif's own integrated development framework (ESP-IDF) attached to a conventional IDE.

B. Single-Board Computer (SBC)

Equally important in the computing side of our project is the use of a single-board computer (SBC) to complement our ESP32 microcontroller. The SBC must host the user interface, which consists of both a local and web interface. It also must communicate bidirectionally with the microcontroller. The same benefits and uses of RISC architectures in microcontrollers also apply to SBCs, As such, we have chosen the Raspberry Pi 4.

The Raspberry Pi 4 is the previous generation of Raspberry Pi's main single board computer. It features a 64-bit quad-core 1.8GHz ARM Cortex-A72 processor paired with either 1GB, 2GB, 4GB, or 8GB of LPDDR4 RAM. Storage is also handled solely by a microSD card slot. Like the Raspberry Pi 5, the Pi 4 has 2.4GHz and 5GHz 802.11ac Wi-Fi with Bluetooth 5.0, as well as Gigabit Ethernet. It also has two micro-HDMI ports, and a 40-pin GPIO header. It requires 5V 3A for power, and lacks the power button of the Pi 5. Compared to the Pi 5, the Pi 4 still provides good performance and lower power consumption at a lower cost. A 4GB version provides better value than the Pi 5, at $55.

C. Current Measurement Hardware

We have implemented the INA260 within our design, specifically, for the per-port electrical monitoring. We chose this device as it provided the necessary tools needed for our project to gather voltage, power, and integrated current measurements while also communicating over the I2C interface. The INA260 also incorporates an internal shunt resistor simplifying implementation overall along with a current sense amplifier which also contains an ADC allowing us to provide accurate measurements when measuring current and voltage.

The INA260 also offers access to real-time data to the ESP32. Essentially, the system is able to view charging and load behavior as well as if any abnormal conditions were to occur within the system itself. [2]

D. Current Switching Hardware

Each of the USB charging channels use the TPS22918 in our design. This is essentially to incorporate independent output control. When dealing with the load switch, it is sent by the ESP32. This aids for power to be delivered to each of the ports whether it is disabled or enabled separately. This is essential towards our device and our project overall because there may be a case where one device will need to still charge (in which normal operating conditions will remain) while a different device may need to stop charging as it is fully charged.

The TPS22918 also aids in improving the electrical behavior of the output stages. Incorporates Quick Output Discharge (QOD) which essentially removes any remaining voltage after a port has been turned off. This ultimately avoids there being floating outputs. This also helps in improving port isolation overall to reduce or even completely avoid any false behaviors that were to occur or take place depending on whether or not the devices are disconnected or connected. [3]

E. 5V Regulators

We designed the 5V Regulators using the LM2670 as proposed in the original Senior Design 1 Report. The LM2670 was chosen due to greater documentation on this device and as well as a lower switching frequency which would facilitate simpler PCB design Once testing and evaluation of the regulator boards were completed, increasing the output voltage of the regulators from 5V to 5.25V by swapping from the fixed 5V variant of the LM2670 to the adjustable variant of the LM2670 to combat against resistive losses encountered on the current switching hardware and Cables. To calculate the required resistance to obtain the desired output, the following equation below was used from the LM2670 datasheet

$$V_{out} = V_{FB}(1 + \frac{R_2}{R_1}) \quad (1)$$

Where V_{FB} is typically 1.21V.

In order to calculate the losses experienced through traces, the following equation was used.

$$V_{out,effective} = V_{out} - I * (R_{trace} + R_{cable} + R_{IC}) \quad (2)$$

Where R_{IC} is the resistances of the INA260 and the TPS22918 load switch.

With these changes, this resulted in an increase of 2% for the efficiency. This can be shown when one analyzes the increase of duty cycle by the equation below.

$$D = \frac{V_{out}}{V_{in}} \quad (3)$$

When increasing the duty cycle, this results in a longer time the switch stays on than off which reduces switching loss in the form of heat.

F. 3.3V Regulator

The MP1584 was initially proposed for our project due to high efficiency and smaller footprint size compared to the LM2670, difficulties with the higher switching frequency resulted in unstable operation. This resulted in swapping to the LM2670 Adjustable variant due to the proven effectiveness during the 5V regulator test. This led to an efficiency loss of 2% which will be more than compensated for by the 2% efficiency increase in our four 5V regulators. Further modifications were made from increasing 3.3V to 3.5V and increasing current from 1A to 1.5A to reduce the number of regulators needed to power both the ESP32 MCU board and the peripheral devices such as the INA260 and TPS22918. The change was made to replace the 2nd regulator with a separate regulator designed to power the Raspberry Pi.

When the switching frequency for this regulator was lowered, this necesites a larger inductor to compensate for the lower $\frac{dV}{dt}$. While this expands the size of the board, it reduces EMI, makes component placing/routing more relaxed, and the longer rise/fall times mean less radiated noise coupling into signal likes such as I2C. The equation below shows the calculation to determine the minimum inductor value possible.

$$L_{min} = \frac{V_{out}(V_{in} - V_{out})}{V_{in} \bullet f_{sw} \bullet \Delta I_L} \quad (4)$$

Finally, to confirm efficiency of the regulators, the equation below was utilized to double check and ensure construction of regulators was done properly.

$$\eta = \frac{P_{out}}{P_{in}} = \frac{V_{out} * I_{out}}{V_{in} * I_{in}} \quad (5)$$

G. Raspberry Pi Regulator (5.25V, 3A)

With the second 3.3V regulator gone, this allowed for the construction of a separate regulator designed to exclusively power the Raspberry Pi. The Raspberry Pi is the most computationally demanding component in the system that sharing a rail with other components would open the device to be vulnerable to voltage drop when the CPU is being utilized to its limits. A dedicated regulator allows for the load transients of the Pi to be isolated from the ESP32 and sensing circuitry.

The increase by 0.25V is to compensate for any resistive losses through the USB-C power cable and connector. With this extra allowance, the regulator is always able to deliver sufficient power to the Raspberry Pi without issue.

$$V_{pi} = V_{out} - I_{Pi} * (R_{trace} - R_{Connector}) \quad (6)$$

With this addition, the device no longer needs an additional AC-DC adapter to power up the Pi itself, reducing bulk of the physical and reducing the points of failure this project can have.

By choosing the same regulator piece for all 3 boards, this reduces the number of unique components needed for the project and allows for a simpler debugging procedure and main components are shared between all boards.

H. Software Stack

The user interface was built using a modified MERN stack. The full stack runs directly on the Raspberry Pi, and this stack consists of React as its frontend, Node.js and Express for the backend, and SQLite for its database. Since our system is hosted locally on the SBC rather than in the cloud, SQLite is a better fit than MongoDB. Raspberry Pi OS provides stable Linux-based support for the Raspberry Pi hardware and allows us to efficiently run our backend services, database, and user interface. The backend communicates to the frontend through APIs. When the user sets a new current threshold and timer value, that request is sent from the frontend to the backend through REST API. Conversely, when the backend is sending live telemetry to the frontend, they are sent through WebSocket API. The backend is also responsible for communicating with the ESP32, which is handled through UART.

I. User Interface

Our system will consist of two different user interfaces, both hosted by and managed on the Raspberry Pi. The Local UI was created using the Tkinter library using Python, and the user can interact with it through the 5 inch LCD screen. It communicates to the backend through REST and Websockets API. The home screen displays

real time charge data to the user and the configuration menu allows the user to adjust a port's threshold and timer settings.

The Website UI is hosted by the Raspberry Pi and can be accessed from any device that's on the same Wi-Fi network. The dashboard serves the same functionality as our local UI, in which it can display real-time data and adjust port threshold and timer settings. The Web UI also displays a live graph that shows the current, voltage, and power over time that a device is receiving. Lastly, it uses the database to record and show the user the last five previous charge sessions for each port on our system.

III. System Overview

M.A.P.S. is powered using a 12V DC input that essentially distributes energy across each of the four USB ports present in our design. Each of the output channels here incorporate a 5V supply path, with a measurement stage sending the information associated with the electrical behavior back to the ESP32, then a load switch stage that is voltage-controlled based on the ESP32's output GPIO pins. Note that there is also a 3.3V rail present as well to power the control electronics.

This system, holistically, acts as a closed-loop charger. This means that the hardware taking in the measurement values, is continuously reporting the current, voltage, and power for each of the individual ports. Then, the ESP32 takes this information and ultimately decides whether or not the port remains active or not. Furthermore, if the device's charging current were to drop below the threshold that the user has selected for the device, the controller is able to disable the corresponding port. Instead of having a system that is continuously charging a device, our design allows for the system to operate more intelligently rather than normal chargers that will continue to charge a device.

To add on, another advantage that we have incorporated in our design is the port-level modularity. Since each of the channels are independently switched and monitored, if an issue were to occur at one port, the entire charger would not shut down; overall improving reliability as well as fault isolation.

The following figure shows our overall system diagram for the project. It can be divided into four main portions as discussed: the interface powered by the Raspberry Pi 4, the current switch hardware, the power delivery hardware, and the microcontroller.

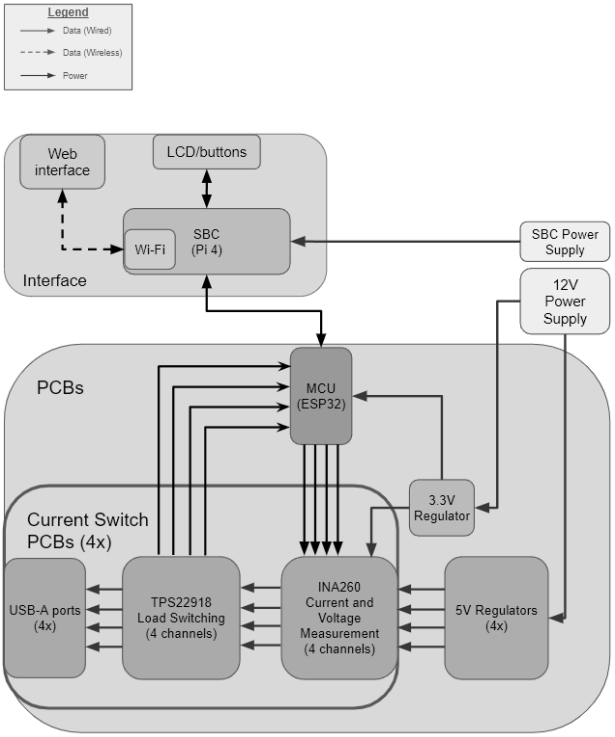

Fig. 2. Overall system block diagram. This illustrates both hardware and software components and how they connect. Arrows indicate flow of power or data (unidirectional or bidirectional).

IV. Hardware Design

A. Power Architecture

Since our product requires three main systems: The Raspberry Pi, The ESP32 MCU board, and The Current Switch Boards, power management for these devices are developed separately which are also based on its voltage and current requirements. As for the Raspberry Pi in specific, it uses a 5 V supply whereas the ESP32 operates from the 3.3 V rail. When dealing with the current switch boards, they have a certain path that is dedicated for the USB power delivery as well as its overall port control also, As previously mentioned, these have been separated. This promotes stability and decreases the interference between the electrical aspects overall within the entirety of the system. Furthermore, fault isolation has also increased between each of the subsystems as well. Also, this helps to simplify how the system is integrated overall. This is done where the power stages and control are operated independently.

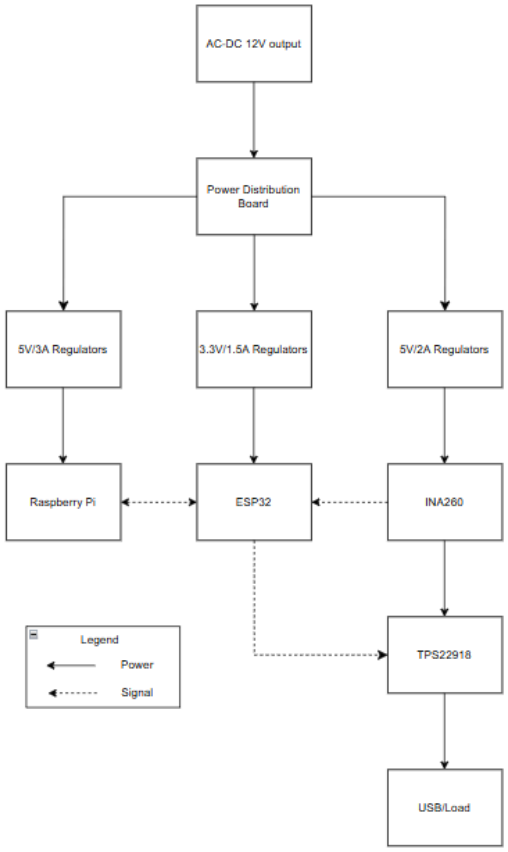

Fig. 3. Overall power architecture block diagram. This illustrates how power is delivered to the various systems present. Due to the nature of the project, signal lines communicating power information to various boards are shown here.

In figure 3, the use of three different regulator types is crucial for the operation of the project as it allows each USB output to have load isolation which allows for the device to operate at max potential despite other changes present in the overall system.

The dashed lines represent the control and telemetry that will interact with the power layer. The ESP32 acts as a central controller where it will receive current/power data from the INA260 via I2C communication. In turn, the ESP 32 will use this information to determine if the load switch should be open or closed. Finally the bi-directional communication allows for the ESP32 to send data over to be displayed on the web site hosted by the Raspberry Pi while the Raspberry Pi can tell the ESP32 what current thresholds are placed from the user to determine when the load switch should be open or closed.

When selecting what the bus voltage should be, a 12V AC-DC adapter was chosen due to the commercial availability of these adapters and the increase in efficiency when compared to other voltage buses.

Finally, the total power budget was calculated during max power operation of all regulatory devices. This event of maximum output from regulators is quite low in probability, it was used as a basis to design the power distribution board to ensure safe operation. For further resilience of high current events, the board was designed to handle a max of 10A before a critical failure event was likely to occur.

B. USB Power Delivery and Switching

Each of the four charging ports incorporates regulated power delivery along with controlled switching. The 5V regulator stage gives the necessary USB output voltage. The TPS22918 allows for the ESP32 to both disconnect as well as connect the load under software control. With this, each of the USB ports have independently managed charging channels.

The port-level switching is vital when it comes to accomplishing the project's battery-health goal. What this means is, based on the measured behavior, specifically current, while a device is charging, the system is designed to reduce current or lessen the charging. The controller is able to essentially "disable" or turn off a device being charged without there being any interruptions with the rest of the devices that are being charged. The load-switch architecture aids in this functionality.

C. Port Monitoring and Feedback

The INA260 allows us to provide measurement-based feedback for each of the USB channels. The current, voltage, and power that is calculated ultimately supplies real-time information that is needed by the control. This also helps in detecting any abnormal behavior that may be present but also supports normal monitoring as well.

The measurement path that is present here is a main piece to our design overall, as it enables the smart charging behavior to our final design.

D. Protection and PCB Considerations

When designing the various PCBs present in the system, considerations of high current loads must be taken in consideration to avoid excess thermal stress on the PCBs. To achieve this, both the top and bottom layer copper are ground layers. Furthermore, the inclusion of via stitching allows these grounds to be connected together allowing for a higher capacity to handle thermal stress. Component placement was carefully considered to avoid heat concentration in singular zones. The equation below

shows how via stitching effectively lowers the the thermal resistance of the board

$$R_{thermal} = \frac{t}{kA} \quad (7)$$

Where k is a constant of 0.048, A is the cross sectional area, and t stands for thickness.

To handle the current going into the boards, increasing trade widths and copper oz. By increasing the trace width, more current was allowed to flow while also reducing the amount of heat generated along that trace.

$$I = k \times (\Delta T)^{0.44} \times A^{0.725} \quad (8)$$

By increasing the copper pour, we allow for a higher current to pass through the copper. To prove this, analyzing A from the equation above, we can derive A from this

$$A = width \times thickness$$

To ensure signal integrity, decoupling capacitors were placed close to the IC power pins to act like a local charge reservoir which can supply current demand until the main supply can reach the desired operating point. Secondly, switching regulators generate high-frequency noise on power rails due to rapid turning on and off of the current. These bypass capacitors can provide a low impedance path for AC noise to return to ground quickly. This allows for the supply voltage to remain stable and avoid corrupting I2C communications. Below shows the equation for decoupling and the relation between how the change in voltage affects the current.

$$I = C\frac{dV}{dt} \quad (9)$$

Once PCB layouts were completed, placement of silkscreen labels for component values were placed to allow for ease of identification during assembly and debugging, consistent via drill sizes to reduce fabrication cost, and standardization of component orientation to simplify the soldering process and inspection.

Finally, test points at key nodes such as input and output, as well as I2C lines allow for inspection of voltages and current to ensure proper connections were made to ensure proper operation. Finally LEDs are placed at major points of success to clearly identify if power is being delivered to key areas.

E. 3-D Printing Enclosure

For our project, we have designed an enclosure model for both the control module and the charger module. This essentially aids in housing all components for the project in a singular housing structure to keep everything in place and put together. The enclosure has been constructed so that wiring can be easily implemented and connected through the system. Spaces, essentially "windows" have been incorporated in the structure as well also for the purpose of airflow and connectivity.

These components incorporate the front UI, rear ports, and an internal layout. For the control module specifically, this will provide an easily accessible UI, specifically, the LCD touchscreen display, a connection zone within the enclosure where each of the USB ports as well as the DC input is designed to limit and reduce interaction with any possible internal wiring. The Raspberry Pi will have a designated place to sit and as for the wiring the design allows for clean wiring overall.

For the design of the charger module, this will incorporate the MCU, all PCBs such as the 3.3V and 5V regulators, USB-A ports, current switches, and all necessary hardware components that go within the device. The model is also designed to incorporate air flow and clean wiring for easy access and debugging.

Overall, the control model enclosure as well as the charger model enclosure play a significant role in the housing structure in our design and system.

V. Software Design

There are two main subsets of our project's software design. Since we have both an ESP32 microcontroller and Raspberry Pi 4 single-board computer, communication and control are imperative for both. The ESP32 takes in the current and voltage values from each INA260 measurement IC over I2C, and must make decisions based on both the measured values and the current thresholds sent to the ESP32 from the Raspberry Pi over UART. The ESP32's outputs are both the GPIO per-port outputs to each TPS22918 load switch, and UART transmission of charging statistics to the Raspberry Pi.

The Raspberry Pi's main job is to facilitate the user interface between a human and the microcontroller, which in turn controls the charging hardware. It takes inputs from both of its interfaces: the local user interface powered by Tkinter, and the web user interface powered by the MERN stack, as well as data from the ESP32.. Those inputs consist of a minimum charge current threshold and recharge timer, which are synced between both interfaces. Additionally, the ESP32 sends it the measured charging data. Its outputs consist of that charging data displayed on the UI, and messages sent back to the ESP32 with the user's configuration settings.

A. Frontend

There are two frontends that have been implemented into our system are the local UI and the website UI. The website frontend of our project was built with React using JavaScript. The website UI allows the user to view the live status of all the USB ports. A user can also send individual port configurations for the current threshold and timer from the UI. The website frontend also shows a live charging graph of current over time, voltage over time, and power over time while a device is charging. Lastly, the website shows the user the charging data from the previous five completed charge sessions from each port.

The local UI is a smaller, more simple UI built using the Tkinter library using Python, The local UI runs on a five inch LCD screen that's connected directly to the Raspberry Pi. The local UI is more limited in comparison to the website UI, but it can still receive and update live charging telemetry and send user port and timer configurations to the backend.

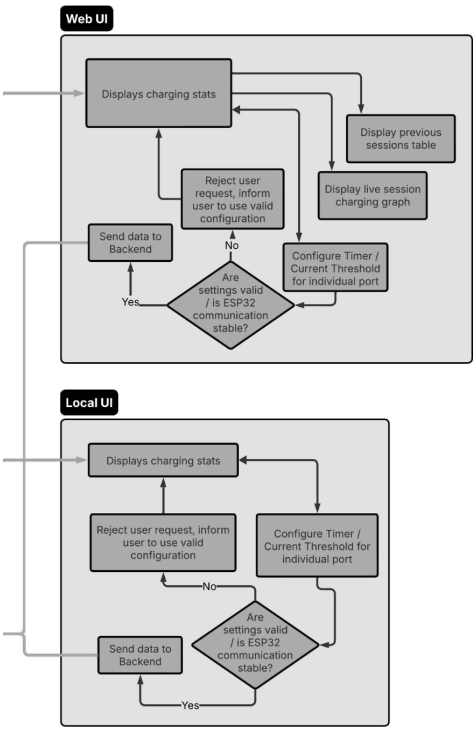

Fig. 4. Software Block diagram. This illustrates the logic within both the local UI frontend and the website frontend and how it connects to the backend

B. Backend

The Node.js and Express backend is what governs most of the logic when it comes to what the user is able to see and interact with. When the user sends a new threshold and timer value, the backend will save those configurations in the SQLite database so that it can store those configurations in case the system gets powered off. The backend also calculates the average amount of current, voltage, power, and energy a device is receiving while it's being charged. When the device transitions to its complete state, it will record those average stats within its database and present them to the user in a table in the web UI.

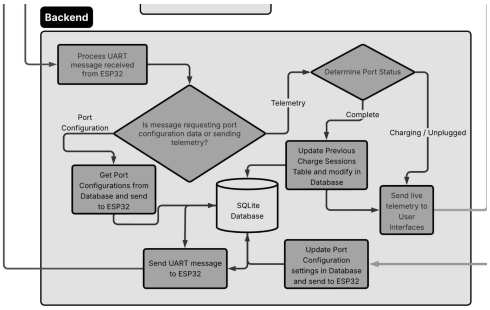

Fig. 3. Software Block diagram. This illustrates the logic within the backend and how it connects to the ESP32 and the frontends.

C. Communication Protocols

Our system's components communicate with each other in several different ways. Between the backend and both frontends, the Raspberry Pi uses a combination of both REST API and WebSocket API. WebSocket API sends real time data from the ESP32 to the user, while REST API sends on-demand port configuration settings from the user to the ESP32. GraphQL API was considered but ultimately omitted because GraphQL would have added unnecessary overhead without providing meaningful benefits over REST API.

The communication from the Raspberry Pi to the ESP32 is handled through UART protocol. The ESP32 is responsible for sending live telemetry data for each port while being capable of receiving data for new port configurations dynamically. The backend is responsible for parsing UART live telemetry data from the ESP32 and transmitting it to both of the frontends through WebSockets API. Conversely, when a user sends port timer and current threshold configurations, both frontends are responsible for sending that data through REST API to the backend. The backend will then process that data and format it into a JSON message to send it to the ESP32 through UART.

When it comes to the ESP32's software, we have chosen to use the FreeRTOS real-time operating system, as our charger is a real-time system that would greatly benefit from its features. FreeRTOS "tasks" are functions that can

be called independently or on a repeating basis. This allows concurrent execution, harnessing both cores of our ESP32. However, with concurrency comes synchronization bugs, so we must ensure any shared resources are protected. FreeRTOS also provides “mutexes” to ensure mutual exclusion of shared resources, which we are also using in our software. Several periodic tasks exist in our software order to check charging stats, make decisions based on charging stats, switch ports on and off, detect if a device is plugged in, and transmit data to/from the Raspberry Pi’s user interface.

VII. Conclusion

Charging devices used semi-frequently or infrequently is a problem that plagues some hobbyists or enthusiasts with many devices who intend to use and preserve them. Our project intends to demonstrate the solution to this problem, creating an embedded and real-time system with the sole purpose of charging multiple of these devices. However, this project can also be applied to other charging situations with multiple devices in professional environments such as schools and other institutions. Thus, we have created a modular charging station to preserve battery health and increase lifespan of various different types of devices, as is illustrated in this paper.

Biography

Hunter Volonino is a 22-year-old computer engineering student graduating with a bachelor’s degree in May of 2026. He is currently searching for possible roles in the fields of his interests: analog circuits and embedded systems.

Sebastian Sotomayor is a 23-year-old computer engineering student graduating with a bachelor’s degree in May of 2026. Later this summer, he will be joining General Dynamics Mission Systems as an Embedded Software Engineer. His interests lie within embedded systems and software development and he plans to further expand his software expertise beyond embedded systems.

Jonathan Luna is a 20-year old electrical engineering student graduating with a bachelor's degree and a minor in mathematics in May of 2026. He is currently accepted into the EE Masters program at UCF and currently searching for possible roles in Control Systems, Radar, and Digital Signal Processing

Siya Sharma is a 22-year-old electrical engineering student graduating with a bachelor’s degree in May 2026. Later this fall, she will be joining Texas Instruments as a Manufacturing Process Engineer. Her interests lie within semiconductor technology and manufacturing, and she is eager to apply her skills to further innovation in the industry.

Acknowledgement

Our team would like to graciously thank our advisor, Dr. Saleem Sahawneh, for the support and expertise he has provided to us over the course of the project for various parts.

We also would like to thank our reviewers, Dr. Sonali Das, Dr. Avra Kundu, Dr. Nazanin Rahnavard and Dr. Azadeh Vosoughi. They have been instrumental in our journey through education and in turn our project.

Additionally, we would like to thank Dr. Arthur Weeks for his support during the debugging and implementation phase of our project.

References

[1] Battery University, “BU-808: How to prolong lithium-based batteries,” Battery University, https://batteryuniversity.com/article/bu-808-how-to-prolong-lithium-based-batteries (accessed Aug. 29, 2025)..

[2] Texas Instruments, “INA260 Precision digital current and power monitor with integrated shunt.”

[3] Texas Instruments, “TPS22918 5.5 V, 2 A, 90 mΩ, 1-channel load switch with quick output discharge.