ANT-WEIGHT BATTLEBOT – DC HAMMER

By

Carson Sprague

Ian Purkis

Gage Gathman

Final Report for ECE 445, Senior Design, Spring 2026

TA: Haocheng Yang

06 May 2026

Project No. 37

Abstract

This report details the design, implementation, and testing/verification of DC-Hammer: an invertible, hammer equipped ant-weight battle bot. The project focused on creating a fully customized robot platform with an emphasis on reliability, responsiveness, and safety. DC-Hammer was designed from the ground up using completely custom hardware and software drivers.

This project combined elements of circuit design, hardware integration, and low-level embedded software to achieve semi-autonomous, intuitive control. Throughout DC-Hammer's development, emphasis was placed on integrating multiple systems for the purpose of balancing offensive and defensive measures, ultimately creating a competitive battle bot. Sensors were used to "close the loop" and provide DC-Hammer with limited deterministic autonomy.

Testing and verification showcased fulfillment of most of the initially outlined design requirements, including reliable wireless connection, accurate and consistent ultrasonic and accelerometer measuring, battery/circuit protection, and stable motor control. Although several requirements were not fulfilled, such as slower-than-expected inversion detection and inconsistent motor rotational displacement, DC-Hammer operated as intended in competition, validating the design choices and steps taken throughout the development process.

Overall, the project demonstrated the successful integration of embedded systems, power electronics, wireless communication, and sensor processing to create a single cohesive platform. The experience of brainstorming, developing, and presenting DC-Hammer, highlighted the value of strong technical and communication skills, cooperation/collaboration, cross-functional work, and system integration.

Contents

1. Introduction

A Battle Bot arena sees a diverse range of weapon systems, physical designs/shapes, and motor platforms. Strategies often vary between defensive and offensive emphases. Our primary goal in designing our Battle Bot was balancing these elements while remaining within the boundaries of competition regulations: namely, weight and material. To achieve this balance of subsystems, we decided to build everything from the ground up. We would not use prebuilt drivers (hardware or software) or IDEs for our control systems and would instead engineer our own lightweight device drivers for each primary component and design our PCB with custom gate drivers. This approach would allow us to trim the fat of our overall design and have an optimized and focused final product. At a high level, our Battle Bot utilizes offensive and defensive measures, while remaining wary of power consumption and current/voltage protection. Our primary goal was in designing a safe and reliable robot that did not cut corners with intuitive control. As we will later detail in our subsystems, we put an emphasis on determinate behavior, synchronization, and safety checks throughout our design to protect onboard systems and the surrounding environment (including bystanders), while also enabling the robot itself with some agency over its own action.

Our Battle Bot, DC-Hammer, utilizes a long “attack” arm, including a hammer head with wedge attachments, as the primary offensive measure. For defense, our robot uses control inversion/adjustment if flipped to continue standard operations. User input and the attack arm’s home position are automatically adjusted when the robot is inverted so that the pilot does not have to change how they interact with the controls. Our robot has eight possible directional inputs, each with variable speed, enabling fine control over the robot’s movements. Additionally, the attack arm is automatically driven by an onboard ultrasonic sensor acting as a proximity detector, with optionality for manual control of the attack arm. We utilized a wireless Bluetooth connection to pilot the robot remotely from a laptop terminal.

Figure 1 – Images of Completed Battle Bot Design

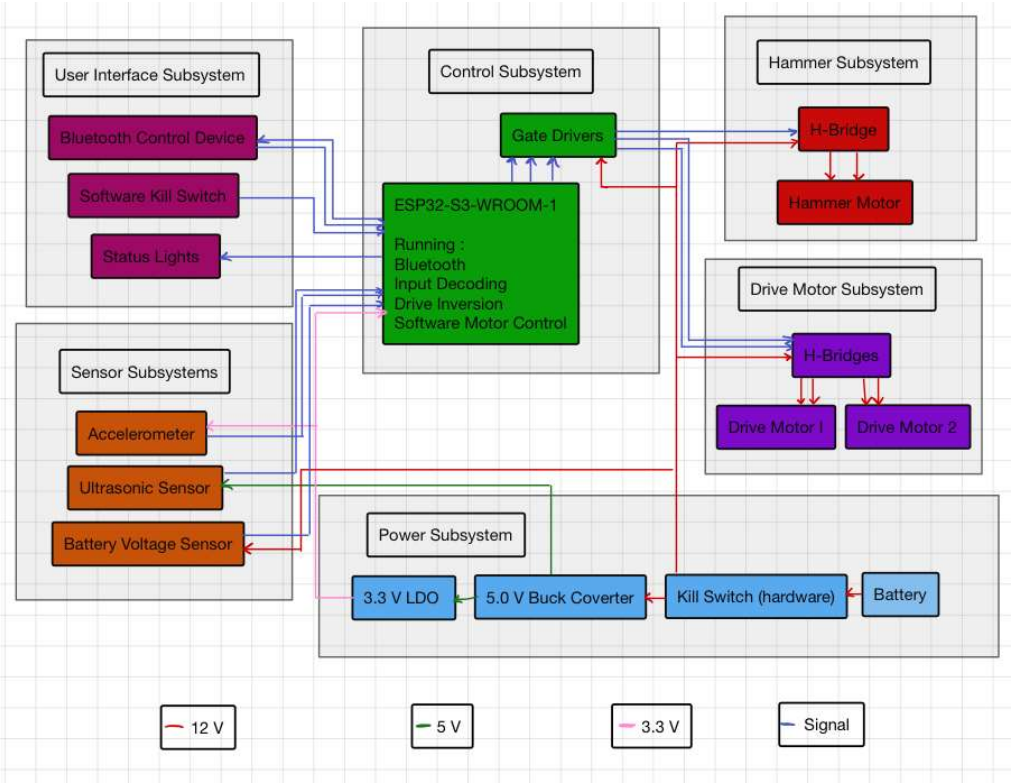

Figure 2 – System Block Diagram

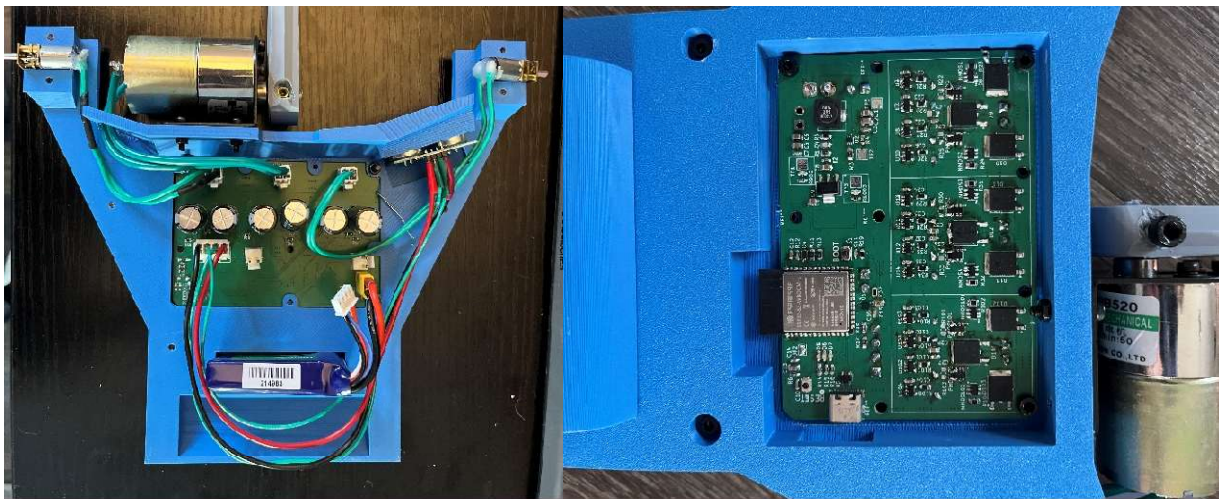

Figure 3 – Images of Robot Internals/PCB

2. Design

2.1 User Interface Subsystem

The user interface subsystem is responsible for providing a seamless interface for the user to control and monitor the battle bot. Users control the bot via a Bluetooth connection with a python script, run on a laptop, and can monitor the bot's status via a status LEDs on the PCB. A kill switch is located on the bot, as required by competition specifications.

2.1.1 Microcontroller

The microcontroller (ESP32-S3-WROOM-1) has built-in Bluetooth support, supporting Bluetooth 5 and BLE. We used the official Espressif Bluetooth Main API [2] for setting up the connection and reading input from the connected Bluetooth device. The MCU runs a loop that processes incoming Bluetooth packets. Along with taking input, the MCU blinks an LED depending on the bot status via GPIO pins.

The MCU interfaces with the ultrasonic sensor, the accelerometer, the battery voltage sensor, and also handles the Bluetooth IO connection. This serves as the heart of our robot, to this extent it is the most important single item. Thus, it is critical it remains powered, and for which it requires a 3.0-3.6 V power supply, capable of 500 mA of current draw to function.

2.1.2 Kill Switch

A physical kill switch is required per competition guidelines, but it is important nonetheless to promote the safety and wellbeing of those nearby. We also provide a kill switch remotely via the keyboard controller script and Bluetooth input handler to further promote safety.

2.2 Sensor Subsystem

The sensor subsystem collects data about the current orientation of the battle bot and position relative to the other battle bot. Namely, an accelerometer is used to keep track of the vertical acceleration of the bot, and an ultrasonic sensor is used to track when a bot is in front of us, in striking range. We aim to

make our control system robust to flips by automatically adjusting based on orientation, and maintain efficient use of the attack arm via automatic triggering.

2.2.1 Accelerometer

We use an MC3416 accelerometer as it provides an interrupt-based flip detection framework. By using an accelerometer and keeping track of the vertical acceleration, we dynamically update our motor control system based on our orientation. This provides a seamless control interface for the user. The MC3416 uses the SPI protocol to transmit data to the MCU (ESP32-S3) via four GPIO pins [4]. We chose to use SPI over say I2C because of the performance benefits, as having up-to-date orientation data is critical for controlling the bot in a competitive environment.

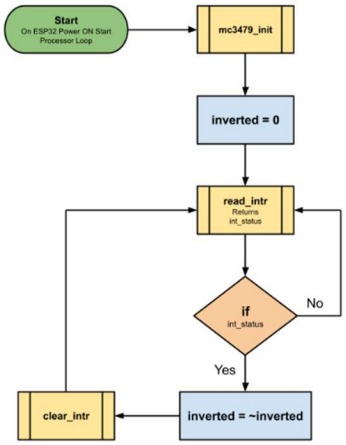

Figure 4 – MC3479 Flip Detection Flow Diagram

During our processor task loop, onboard the ESP32, we read the interrupt register of the MC3479 and mark an inversion flag if this is triggered. In our initial configuration of the accelerometer, we activate a flip threshold interrupt condition, setting the threshold value and delay between measurement recorded and interrupt raised. Additionally, we set the interrupt register to reset when the condition has ended. This turns the interrupt flag from just an event to an inversion state, which we utilize to adjust user control and hammer positioning

2.2.2 Ultrasonic Sensor

We use an Adafruit 4007 ultrasonic sensor not only because it should be satisfactory in terms of getting the distance to the nearest object in front of us, but also because it is functional with both 5v and 3.3v [1], which gave us more flexibility in our power subsystem design. This sensor, like most other ultrasonic sensors, transmits data via two GPIO pins, sending a trig and echo signal. To calculate distance to an object in front of us, we fire the ultrasonic sensor and then use the roundtrip time to calculate the distance to said object in front of us (a rival battle bot). Given that sound travels at around 343 m/s, or more useful to us, 0.034cm/microsecond, we can use the following equation to calculate the distance in software.

$$D = \frac{T * 0.034}{2}$$

In the above equation, D is in centimeters, and T is in microseconds. We had a distance threshold of 25cm, which is the limit in which the hammer automatically would automatically strike because the object was in range.

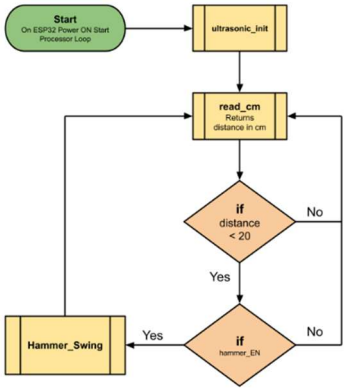

Figure 5 – Ultrasonic Measurement Flow Diagram

In our processor task loop, onboard the ESP32, we simply trigger the ultrasonic and read the current measurement. If this value falls within a particular threshold we call the arm swing protocol.

2.2.3 Battery Voltage Sensor

The battery voltage sensor simply reads in the voltage being supplied to the chip as a digital value via one of the GPIO pins. If at any point the voltage we read in dips below 9 volts, we shut down the system. This is a protective measure to ensure the LiPo batteries don't become over-discharged. We use a simple equation to calculate the real voltage from the digital value which we found from comparing the voltage supplied via a DC power supply to the digital value we read on the MCU, where V is the real voltage, and X is the digital value read in.

$$V = 0.024 * X - 26$$

2.2.4 Gate Drivers

We have chosen DGD0211CWT-7 as gate drivers. These chips will be able to take our final PWM signals at 20 kHz at 3.3 V and apply the battery voltage ~11 V to the gates of the H-bridge MOSFETs. These devices serve a critical role in the control circuit, bridging the gap between the "low" and "high" voltage segments.

2.3 Hammer/Wedge Subsystem

The primary offensive tool is the hammer/wedge "attack arm". This arm is driven by a high torque motor and activated by a proximity-detecting ultrasonic sensor for faster than human reaction time

when opponents enter attack range. The attack swings also have a user input trigger coming from the 'E' key on the laptop controlling the bot.

2.3.1 Hammer

Our primary attack, and namesake of the robot, is the hammer. The resting position of the arm lies approximately 90 degrees from the plane parallel with the robot chasse frame. We wanted this angle to be as wide as possible while still maintaining balanced weight distribution around the body of the bot. Considering the weight of the motor, and the shape of the armor shell, 90 degrees was the optimal resting position to prevent the arm from interfering with its shell when returning to its resting position and maintaining balance. When the hammer swing protocol is triggered, we use a 95% duty cycle forward PWM signal for 300 milliseconds. Based on the estimated maximum speed of our motor 300 milliseconds should account for approximately a quarter of a rotation: 90 degrees. We then send a low or zero PWM signal to stop the motor for a small delay to halt the motor. This prevents shoot through on the H-bridge motor driver when the backward or rehome motion signal is sent to the motor.

2.3.2 Wedge – Removed Functionality

Our original goal was to include a full secondary attack mode: the wedge. When activated, by user input, the resting position of the arm would change to be parallel with the plane of the chasse frame. When triggered, the arm would lift with the goal to flip over and disable rival Battle Bots. There were two main reasons why we removed this function.

The first was the size of our data packet. Each BLE packet sent from the control device to the ESP32 is a single byte. Utilizing the version of BLE connection we had built out, we were limited to those 8 bits. Our controller design used each bit as a state for a variety of keys with one of the bits being a state transition tracker. This meant we only had 7 bits to use for keys. Four direction keys, a speed-up or "sprint" key, manual arm activation, and our safety remote kill switch occupied the 7 essential buttons. This left our "change attack mode" key out. We considered using a key combination to trigger this swap but were concerned about race conditions relating to the keys used in the combination. Suppose we made the forward and backward key combination (W and S) the trigger for a mode swap, the slight human difference in timing of the two required keystrokes could mean the robot would move one of those directions, briefly, when the pilot did not want to move the robot.

Additionally, when in a dedicated wedge mode, the ultrasonic sensor becomes defunct as the presence of the arm in front of the bot will continually trigger the threshold measurement. This would mean disabling the sensor during wedge mode. Working with a brushed DC motor, there was a certain, and acceptable, amount of unpredictability with the amount of rotation performed by the motor given a constant/regular PWM signal for a particular time interval. This meant that a large portion of the software system revolved around synchronization and locking with virtual motor resources. Dedicated wedge mode added additional complexity to this system, with the potential for locking out other core functionality of the robot. The overhead of adding this to our system was deemed too long of a task to get right within our development window.

2.3.3 High Torque Motor

We used a 12 V, 45 RPM, brushed DC geared motor, with a rated stall torque of 50 kg-cm. This motor conducts “smashing” rotations of the hammer. For this purpose, this motor requires 0.68 to 2.19 A of current, rated and stall respectively. In combination with the drive motors, this compromises the majority of the current required by and weight of our robot. As we neared completion of our development process, we realized that this motor, while forceful, was not very fast. Even at top speed, this motor takes more than a full second to complete a single rotation. In future iterations, using a higher speed motor for the hammer would allow for quicker, more accurate, and higher damage attacks.

2.4 Power Subsystem

To give life to our battle bot, a power system is needed to supply necessary power to all components, including drive motors, hammer motor, MCU, sensors, etc. A variety of components are used to ensure a safe and robust system to deliver power where necessary.

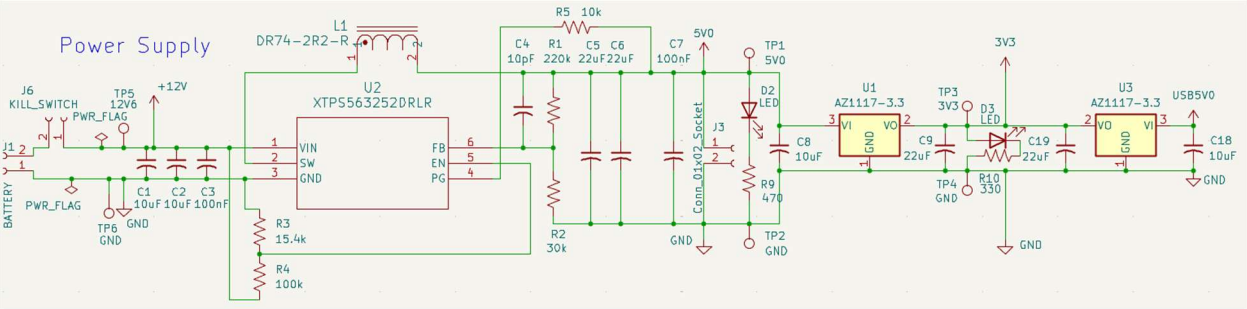

Figure 6 – Schematic of Power System (Right side 3.3V LDO only for Testing)

2.4.1 Battery

The main battery is to provide requisite power for the entire system. For this purpose, a 3s (11.1v) LiPo battery of sufficient capacity is required. This battery connects directly to the kill switch, through which there are connections to all major components, particularly the 5V Buck Converter and to the motors. This is a critical component due to the fact that it must provide sufficient current and voltage to three 12v DC motors which represent the majority of our battle bot’s power consumption. Particularly, this battery must be able to store sufficient energy to power these motors for 3-minute bouts.

To this end, we upsized our original 450 mAh batteries to 800 mAh batteries for a greater margin. In the duration of our testing, none of the batteries were discharged below 80% capacity. More directly, we decided our batteries should operate between 4.15 V and 3.3V, both within the margins of safe operation of these batteries by a margin. During the multiple test operations and bouts which we conducted, our battery voltage when set to recharge never strayed below 3.95 V. This demonstrates that we had a large margin of extra battery space.

2.4.2 Buck Converter – 5.0 V

We have conducted a redesign since our proposal. After subsequent thermal capacity calculations, it was obvious to see that our original LDO scheme, even under resistive paralleling, would pose significant risk of issues. Instead, after consideration, we instead decided to leverage existing integrated MOSFET buck converter IC. To this extent, we decided to leverage TI’s TPS563252DRLR, which is a 3-amp rated

94% efficient, low resistance buck converter. This was configured to have less than 3% voltage ripple output directly at 5V.

2.4.3 Voltage Regulator – 3.3 V

Multiple components on our battle bot will require a 3.3 V power supply. Principally, the MCU, an ESP32-S3 chip, will require a rated 3.3 V and 0.5 A of power to operate with a minimum and maximum operating voltage of 3.0 V and 3.6 V respectively. To this constraint, we have chosen the AZ1117C-3.3, whose datasheet indicates a ±1% accuracy, minimal regulation, and a minimum of 1 amp of load.

2.5 Software Subsystems

Given the complexity of the design, we deem it necessary to describe the software subsystem separately. The software subsystem makes it possible for the bot to react to stimulus, whether from keyboard input or environmental changes read from the sensors. All the code was written in C using the Espressif IDF Framework. We chose to go this route instead of using the Arduino framework because of the control that was offered and the performance that we could leverage from that control. For example, we were able to take full advantage of both cores on the system, which we otherwise would not have been able to do. The figure below breaks down the overall software architecture, but in summary, we dedicate one core to an event processing loop which reads in sensor data and updates global variables, while the other core handles Bluetooth connection events and incoming keycode control packets.

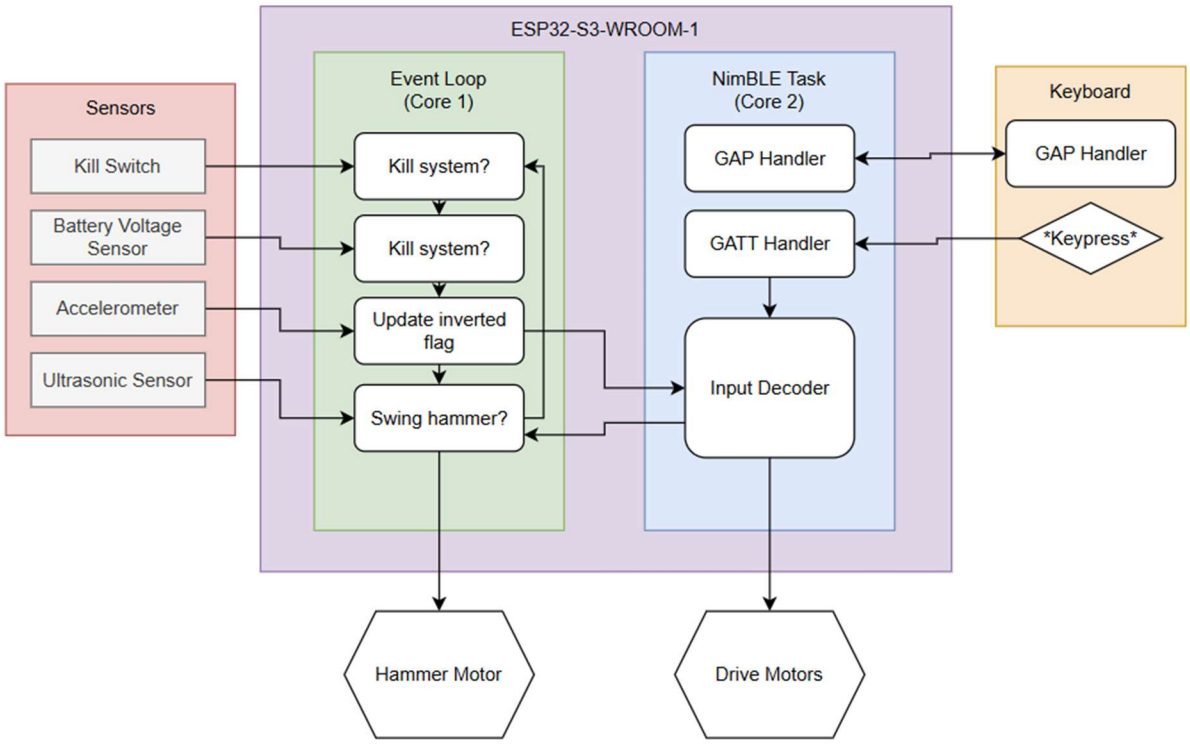

Figure 7 – Software Architecture

2.5.1 Bluetooth

One of the core functionality features that was required for our battle bot is wireless control. We leveraged the ESP32-S3's Bluetooth (Low Energy) capabilities to achieve this. Specifically, we utilized the Apache Newt NimBLE Bluetooth host stack, which the ESP-IDF supports [7]. We chose to go down this route rather than a traditional Bluetooth stack because of the lower RAM footprint, event-driven design, suitability for resource constrained systems, and that it integrates well with FreeRTOS (which runs on the MCU).

The bulk of the code went into initializing the Bluetooth LE stack, defining interfaces, and registering for subscriptions that go along with that. Core to BLE functionality are the Generic Access Profile (GAP) and the Generic Attribute Profile (GATT) [6]. These act as interfaces, where GAP describes how the device behaves during scanning and connection events, and GATT defines what data is accessible to clients and how (read, write, etc). In our case, the ESP32 is a server to the Bluetooth keyboard, and initializes its own GATT table. Aside from being mostly pedantic system initialization, these interfaces allow us to define and register our own callbacks when an event or notification occurs (hence event-driven design), instead of constantly polling for new data, which wastes CPU cycles and drains power.

The GAP callback is responsible for decoding the GAP event that the MCU just received and handling it accordingly. It will fire when the ESP32 connects or disconnects to the Bluetooth keyboard. These events and the Generic Access Profile as a whole are responsible for just setting up the BLE connection. The event will be one of the following and will be handled as such [7].

```
BLE_GAP_EVENT_CONNECT

BLE_GAP_EVENT_DISCONNECT

BLE_GAP_EVENT_CONN_UPDATE
```

The GATT callback is more interesting as the GATT notifications contains the key/control info from the Bluetooth keyboard. This callback fires when the GATT client (keyboard) sends data to the GATT server (ESP32) containing the current state of one or more keys. The input decoder will then be responsible for parsing the pressed/released keys and driving the motors accordingly.

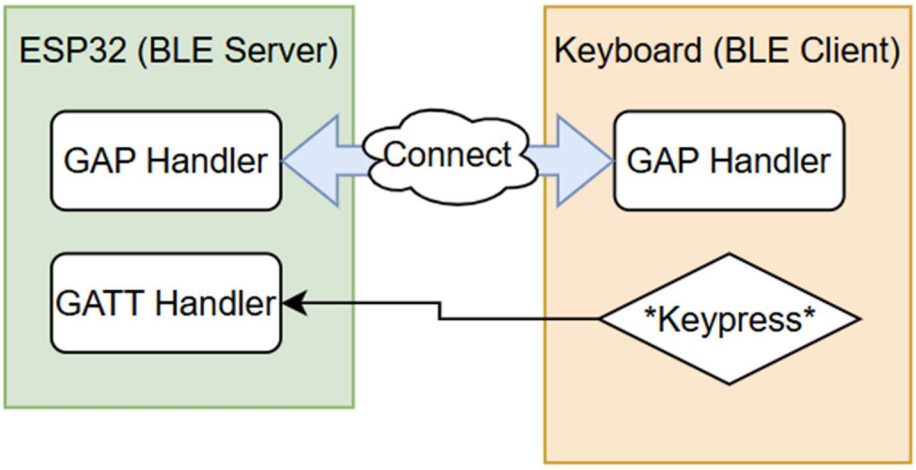

Figure 8 – BLE Client/Server Interaction

2.5.2 Input Decoder

The input decoder is synonymous with our GATT callback handler as referenced in section 2.5.1, since the callback is triggered whenever a new keycode packet comes in. The decoder is responsible ultimately for driving the motors, while also considering the current position of the bot (upright or inverted).

One challenge we faced in designing the decoder was handling multiple keypresses at once, say SHIFT and W. We need to handle that combination to enable a 'sprint' mode, but receiving each keypress as a traditional ascii code wouldn't allow for that. To get around that we developed a custom keycode packet which enabled us to track the states of only certain keys, as shown below.

W	A	S	D	SHIFT	TAB	SPACE	DUP

Key	**Function**
W	Forwards
A	Left
S	Backwards
D	Right
Shift Modifier	'Sprint'
Space	Hammer
Tab	Change Mode
DUP	Duplicate Packet

Figure 9 – Custom Keycode Packet

Since we chose to keep track of only certain control keys, we were able to fit our packet in a single byte. This allowed us to keep our BLE packets extremely small and make the input decoder extremely fast by

just performing bitwise operations (roughly 500 microseconds as referenced in section 3). Furthermore, we were able to utilize a duplicate flag (least significant bit) which enabled us to exit processing early if there were no state changes. This was also useful for preventing h-bridge shoot through, which is covered in section 2.5.3.

The only other signal relevant to the input decoder comes from the accelerometer. A global variable describes whether the bot is upright or inverted and is used for driving the motors correctly depending on orientation, because driving the motors forward in an upright position may be clockwise, but in an inverted position would be counterclockwise.

2.5.3 Software Motor Control

Now that we've collected input from the user, and all the signals have been handled, we can finally drive the motors. To handle motor control, we used Espressif's MCPWM library [5] which generates PWM signals for motors.

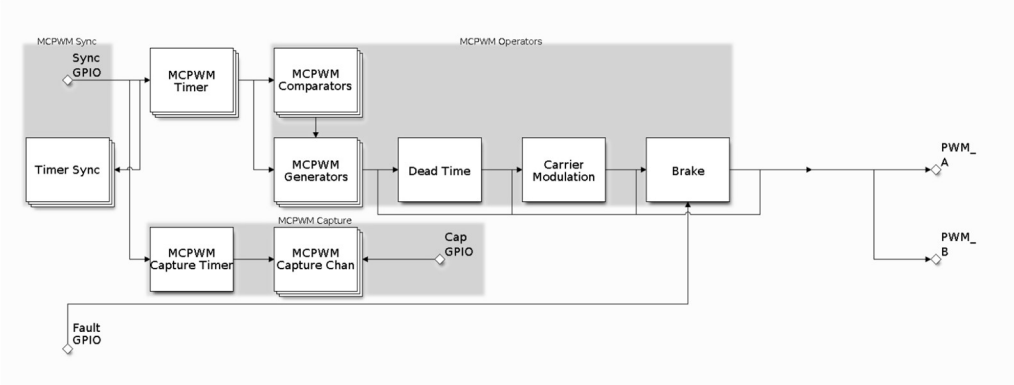

Figure 10 – MCPWM Recommended Configuration Diagram [5]

This library requires configuring several modules for each motor to drive the PWM signals via the GPIO pins of the ESP32. We used the timer, comparator, operator, and two generator modules for each motor. The timer and comparator modules take the number of ticks in each period of the input duty cycle and step function to build the PWM signal timing. The operator acts as the connective tissue between the modules and the generators output the PWM signal to the GPIO pins. We utilize two generators for each motor in order to have a distinct forward and backward direction, with each generator signal driving an individual GPIO pin: one for forward and one for backward. This separation allows for finer, more intentional motor control at the expense of having to implement dead time ourselves, as the ESP32 does not have enough resources to support six different deadtime configurations.

Our solution to implementing dead-time was to track state transition on incoming keyboard inputs. For fixed action protocol, like hammer swings, we simply hard coded a dead-time period between changing the motor's input direction. With driving control, these changes in direction are not as fixed or predictable. Considering the total combination of direction input we could potentially receive we figured that holding the motors low for a dead period on any change in directional input, or inversion, was the

best method to protecting the motor. The least significant bit of our BLE packet is a state transition bit. The laptop Bluetooth script compares the current packaged packet to the previously sent packet and marks the transition bit accordingly. In the input handler we check this bit of the incoming data and an inversion change flag. If either of these flags reflect that a change has occurred in the input handler's state we activate a dead time period where the drive motors are fed a low signal for a two millisecond wait period. This prevents shoot through by temporarily deactivating the motors between any two events that could potentially change the direction of the motor.

Utilizing the MCPWM library afforded us flexibility in configuring and driving our motors. Our custom driver/API, built on the framework of MCPWM, allowed us to tune our parameters in software to achieve whatever speed we deem fit for competition. This also allows us to support a crawl, walk, and sprint speed if we find it advantageous. To handle veering left and right, we increase the duty cycle of one motor while decreasing the duty of the other. To achieve tank turning, we drive one motor forward and the other backward. We input individual direction controls, based on key states, to a drive motor function that checks the combination of inputs and activates left and right motors accordingly.

2.6 Drive Subsystem

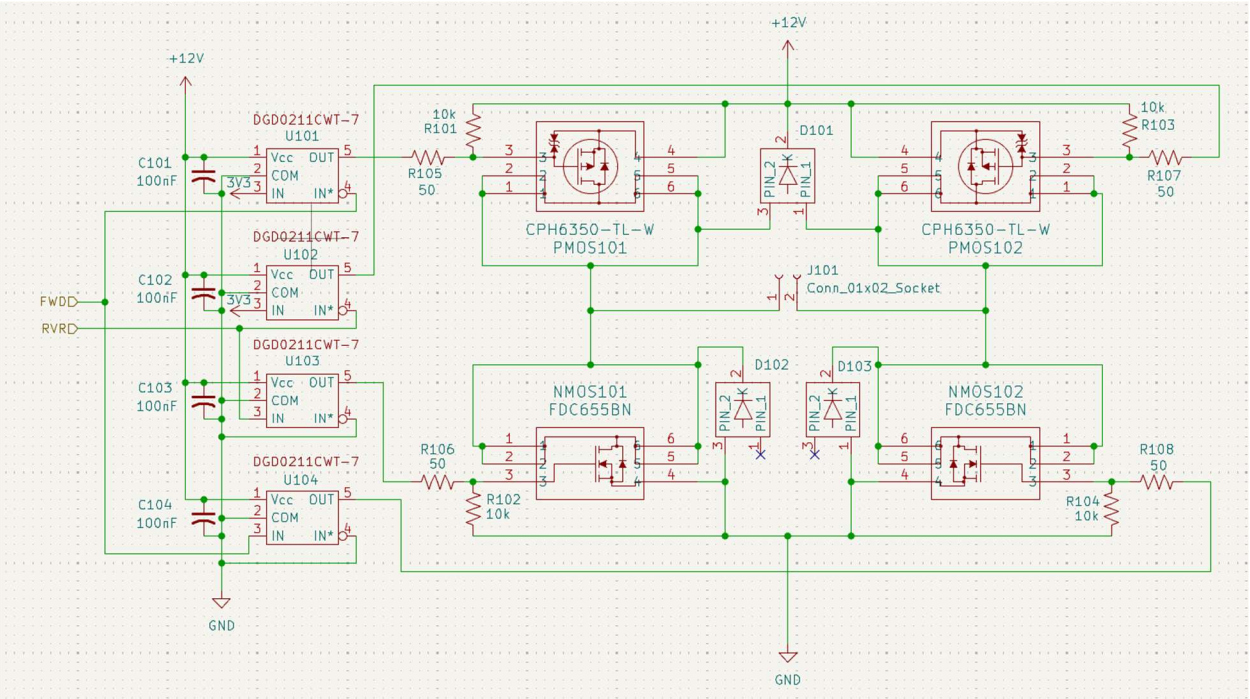

Figure 11 – Canonical Structure for our H-Bridges

2.6.1 H-Bridge NMOSFETs

For our motor to operate properly we will need high-power MOSFETs for our H-Bridge. Instead of purchasing integrated ICs, we have designed an H-Bridge that can be triplicated across all three motors. The chosen devices are as follows, for NMOSFETs Onsemi's FDC655BN, which are 30 V, 6.3 A rated MOSFETs, for PMOSFETs CPH6350-TL-W, which are 30 V 6.0 A rated, both with low on-state resistances and high switching frequency capabilities, for our diode protection we have chosen V20PWM63C-M3/I, which are 60 V 10 A, 0.66 V forward voltage diodes, which should provide current paths while our

MOSFETs are switching. These should be overrated for our application to ensure safety in our circuit. If one of these devices were to fail, it would not only constitute a robot failure but may be critical to battery safety. However, since these devices are overrated, they should not be a limiting element in our design. Like any other critical path electrical component, failure will be caused by loss of voltage, or improper operation.

2.6.2 Drive Motors

Our main two drive motors are 12 V, 600 RPM, brushed DC geared motors. These will propel our battle bot, which is a critical feature for obvious reasons. Their current requirements are 10-160 mA for the no-load to stall current operating range. This helps us inform how we must compute our battery size. These will interface with the H-bridge modules and the ground. These motors will need to be durable, as this robot will likely experience high-g forces being tossed around.

3. Requirements & Verification

Requirements	Verification	Results
Activation of the physical kill switch should disable all motor outputs within 500ms	We will create a testbench like system and run the motors at speed. At that point we will trigger the kill switch and measure the time from switch activation to the motors being disabled.	PASS – The motors lost power nearly immediately and stopped spinning in under 500ms.
The LEDs should display all relevant Bluetooth states to the driver, including Not connected, Connected, and Fault.	We will connect, disconnect, and shut off the Bluetooth keyboard multiple times to ensure the LEDs update the status correctly and quickly to the driver.	PASS – We were able to disconnect and reconnect to the bot multiple times without issue (and see status via LEDs).
The system shall detect with the bot is inverted and apply motor control state changes within 100ms.	We will place the bot upright and log the accelerometer vertical axis value. We will then flip the bot and measure how long it takes for motor controls to update. We will output logging data to the terminal.	FAIL – We could not detect inversion from the accelerometer in under 1 second. We attribute this to the accelerometer chip taking quite awhile to raise the inversion interrupt flag.
The ultrasonic sensor shall detect objects within 5-50cm with an accuracy of +-2cm.	We will place the bot down and place another object in front of it (measured with a ruler). We will log the ultrasonic sensor data to the terminal and compute error.	PASS – We could detect an object with an accuracy of +- 1cm at worst but were often much closer to 0.1cm.
The battery voltage sensor shall disable motor outputs when the voltage falls below 9 volts.	We will replace the battery with a bench supply and slowly lower the voltage, confirming that the motors disable at 9 volts +- 0.2 volts.	PASS – Motor outputs were disabled at almost exactly 9 volts, well within the 0.2 volt range.

The gate drivers shall provide at or near battery voltage gate voltage when the battery voltage is normal (9-12.6V).	We will prove the MOSFET gate relative to the source, drive the PWM at 50%, and confirm the gate voltage is over 10 volts.	PASS – Output Voltage Dependent on Current – Does not drop below -0.5 V below battery voltage.
Hammer default position will change based on the orientation of the robot. After 1 sec of inversion beyond 90 degrees, the default arm position should invert to account for orientation change.	A lot of the expected hammer behavior is the result of multiple sub modules working as intended (i.e. sensor trigger/accuracy, motor control, Bluetooth connection, etc.). To test the rotation threshold for the robot, we could output accelerometer readings to a display (perhaps via Bluetooth to a window on piloting laptop). This would help us see that the inversion is triggered after exceeding greater than 90 degrees of inversion. We can also test the delay in the same way, by inverting the bot beyond the threshold and observing the arm position adjustment within a particular time delay (1 second as of now).	FAIL – Our average period between inversion and arm position adjustment was greater than 1 second. Built into our inversion detection is some delay, about 500 ms, to prevent "over-eager" inversion detection. When testing the accelerometer by itself we also noticed that the time-to-detection was regularly greater 1 second. Additionally, the speed of the motor means that our arm takes a minimum of about 600 ms to rotate 180 degrees. These delays all together meant that our rehome regularly took longer than 1 second to complete. As such our average time from inversion to completed rehome was about 1.5 seconds.
Hammer/wedge swing will be triggered by sensor input when object is within range of the hammer arm. Hammer/wedge swing will be triggered by user/keyboard input	As in the previous design requirement, Success of the swing triggers is the result of multiple subsystems working correctly. Testing this specific behavior is as simple as observing a swing as a result of user/keyboard input, and moving a detectable object closer and closer to the ultrasonic sensor until a swing is triggered. Using a ruler or meter stick we can check at what distance the swing is triggered. Currently this distance is unknown, as we intend to try several variations of arm design and will depend on the arm's pivot point relative to the rest of the robot's chasse.	PASS – We consistently displayed the ability to trigger the hammer swing via keyboard/pilot input and utilizing measurement thresholds via the ultrasonic sensor.

Hammer arm will return to initial position (after swinging action) within +/- 2 degrees	We can utilize the user input for discrete testing of hammer swing rotation. The model of motor we are planning on utilizing does not have the ability to measure rotation. Rotational control would be relative to motor speed (input current magnitude) and duration of activation (input current duration). To verify the actual rotation amount, we could use a protractor and manual swing to observe the correct rotations (forward and backward swing).	FAIL – As previously mentioned, using brushed DC-motors is not ideal for fine or discrete motor control. We can not control an amount of rotation but can attempt to by utilizing the combination of duty cycle of PWM signal over a specific period of time. The performance of the high-torque motor was more variable than we had hoped, even when fed the same duty cycle over the same period of time. Our average variability from the intended resting position was near 10 degrees.
Battery must be able to provide for sustained operation of Battlebot for the duration of one bout of at least 3 minutes in length.	We will conduct test bouts with our robot, activating all subsystems frequently and continuously in the case of motors to validate our loading calculations. Assessing battery voltage before, during, and after using datalogging on our ESP32-S3.	PASS – As Previously Stated our Final battery Voltage Tested with the more accurate battery charger, we ready a discharge over multiple 3-minute bout equivalents on a single charge and did not deplete more than 20% of battery capacity.
5.0 Volt Regulator will need to be able to output consistent load in line with supply for subsystems requiring lower-than-battery voltage.	During standard Current operations we will assess the voltage ripple of our circuit, to ensure it is within the 3% spec in voltage tolerance via oscilloscope.	PASS – Our final voltage output peak was 5.10 V and the minimum was 4.96 V, yielding 2.8% ripple, within our spec.
3.3 Volt Regulator will need to consistently supply a reliable 0.5 amp at steady-state to supply our ESP32-S3 for its operations.	To test this, we will detail our overall performance, whether the ESP32-S3 has consistent current, and whether there is significant thermal load on the LDO.	PASS – Our device never overheated or had the voltage drop out under normal conditions. LDO was warm to touch, but never hot during any testing.
Must achieve a latency of under 1ms from receiving a key press to driving the motors	When a key press is received, it will trigger the GATT handler. We will start a timer in this handler and then stop the timer once we update the duty cycle in the motor	PASS – Our latency was around 0.5ms, nearly half of the time we wanted to achieve.

Note that this is not the time from physical keypress, but the time from which the signal is received at the MCU.	control handler. We will write this to the terminal.	
Spamming keypresses while driving should not cause the bot to lock up or have controls 'lag' behind. In competition we will be pushing our bot to the limits. We cannot have our control system breaking down or lagging under high strain - otherwise we are at a massive disadvantage.	We will alternate through WASD as fast as we can on the keyboard for 5 or so seconds, then we'll stop. If the bot continues driving (meaning our handler is too slow), or the control becomes seemingly random (signals are being corrupted), we need to streamline our software to be more efficient.	PASS – We had no lag issues even after spamming keypresses for multiple seconds (and had no issues during competition).
H-Bridge functionality will be determined by its ability to properly control the motor connected to it, and effectively handle the heat generated.	During current testing of the motors, overall current was assessed, and thermal loading tested to ensure no components failed. Voltage Waveforms were recorded to visualize voltage across motor output, checking for no shoot-through.	PASS – Thermal Loading was very low, to the extent no significant rise in temperature was observed during operations of all three motors. This is likely due to the low current observed. Voltage waveforms showed no shoot-through, which was eliminated as a possibility in code with a 5 ms lock out.
Motors will need to be functional and highly reliable for long durations and under stall/stuck rotor conditions.	1 - To test these parameters, we will conduct basic bench tests to ensure reliability. 2 - Battles will show the durability and performance of our machines.	1 – PASS – The motors were within their current specs and ran to expectations, within reason. However, their operating speed was slightly lower than expected. 2 – FAIL – Our drive motors failed to withstand full battle duration. The geared portion ended up breaking, causing the gears to disengage and lose drive. This could have been prevented with better motor choice.

4. Cost & Schedule

The table below outlines the cost for all the major individual components we plan on using, with a total of $101.69 before tax and shipping. Assuming shipping adds about 5% and sales tax adds another 10%, we can expect to pay just around $120 for components. As ECE graduates from Illinois, we can expect a salary of about $40/hr. Thus, we can calculate the labor cost to be about $40/hr * 2.5 * 40 hrs, roughly $4,000 per team member. With three team members, that's $12,000 in labor costs. The total cost of this project comes out to be just about $12,120.

Component Functionality	Component ID	Link	Price per component	Quantity	Total Cost
PMOSFET	CPH6350-TL-W	https://www.digikey.com/en/products/detail/onsemi/CPH6350-TL-W/4847613	0.71	6	4.26
NMOSFET	FDC655BN	https://www.digikey.com/en/products/detail/onsemi/FDC655BN/979810	0.74	6	4.44
Diodes	V20PWM63C-M3/I	https://www.digikey.com/en/products/detail/vishay-general-semiconductor-diodes-division/V20PWM63C-M3-I/16683117	1.93	12	23.16
Accelerometer	MC3479	https://www.digikey.com/en/products/detail/memsic-inc/MC3479/15292802	1.11	1	1.11
Buck Converter IC	TPS563252DRLR	https://www.digikey.com/en/products/detail/texas-instruments/TPS563252DRLR/20415402?s=N4IgTCBcDaICoAUDKBWAbAZjCsARASgDL4gC6AvkA	0.49	1	0.49
Buck Converter Inductor	DR74-2R2	https://www.digikey.com/en/products/detail/eaton-electronics-division/DR74-2R2-R/667219	0.57	1	0.57
Main MCU	ESP32-S3-WROOM-1	https://www.digikey.com/en/products/detail/espressif-systems/ESP32-S3-WROOM-1-N8/15200089	5.49	1	5.49
Ultrasonic Sensor	Adafruit 4007	https://www.digikey.com/en/products/detail/adafruit-industries-llc/4007/9857020	3.95	1	3.95
Gate Drivers	DGD0211CWT-7	https://www.digikey.com/en/products/detail/diodes-incorporated/DGD0211CWT-7/12702560	0.55	12	6.6
Hammer Motor	12V 50RPM 694 oz-in Brushed DC Motor	https://www.robotshop.com/products/12v-50rpm-694-oz-in-brushed-dc-motor?pr_prod_strat=e5_desc&pr_rec_id=1e033aea8&pr_rec_pid=7487305908385&pr_ref_pid=7487308398753&pr_seq=uniform	15.88	1	15.88
Wheel Motors	12V 600RPM 50:1 Gearmotor	https://www.robotshop.com/products/dyna-engine-12mm-diameter-501-micro-metal-gearmotor-12v-600rpm	6.95	2	13.9
3V3 Regulator	AZ1117CH-3.3TRG	https://www.digikey.com/en/products/detail/diodes-incorporated/AZ1117CH-3-3TRG1/4470985	0.16	1	0.16
Battery	Turnigy 450mAh	https://hobbyking.com/en_us/turnigy-nano-tech-plus-450mah-3s-70c-w-xt30.html	6.39	2	12.78

	3S 70 LiPo pack				
Connectors (Assorted)	JST-VH series	-	0.25	10	2.5
Resistor (Assorted)	Varying Sizes	-	-	-	5
Capacitors (Ceramic)	Varying Sizes	-	-	-	5
Capacitors (electrolytic)	UVR1H102MHD	https://www.digikey.com/en/products/detail/nichicon/UVR1H102MHD/588852	6	1.13	6.78

We aimed to follow the schedule below to stay on track but understand that this is just an estimate and some things bled into other times.

Week	Tasks
Feb 22 – Feb 28 **PCB Round 1 – Feb 26**	Carson: Began work on setting up the build system (Platform.io), Bluetooth code, and design doc Gage: Gage Finish a preliminary board for power rail and H-bridge verification Ian: Assisted Carson with Bluetooth code and researched arm design/length for maximizing impact force
Mar 1 – Mar 7 **PCB Round 2 – Mar 5**	Carson: Continued work on Bluetooth code and prep for presentation. Gage: Create an initial complete schematic for design review. Ian: Continued work on Bluetooth code and prep for presentation
Mar 8 – Mar 14 **PCB Round 3 – Mar 12**	Carson: Completed Bluetooth code / Began input decoder and motor control code. Gage: Complete initial PCB Ian: Complete Bluetooth handler code, wrote python script for establishing BLE connection with ESP32
Mar 15 – Mar 21 *Spring Break*	Carson: Traveling Gage: Staying Home Ian: Traveling
Mar 22 – Mar 28 **PCB Round 4 – Mar 26**	Carson: Continued input decoder and motor control code (LEDC, later replaced with MCPWM). Gage: Validate H-Bridge and Power Rail PCBs. Ian: Wrote Ultrasonic device driver and tested sensor functionality
Mar 29 – Apr 4	Carson: Completed input decoder, ADC voltage sensor code, and helped with ultrasonic sensor code. Gage: Testing PCBs

	Ian: Wrote Accelerometer device driver and data transfer API (SPI interface). Tuned and tested accelerometer functions. Tested current software systems on PCB.
Apr 5 – Apr 11 **Progress Demo**	Carson: Helped with motor control code, inversion, and integration. Got into working state for demo. Began CAD work for 3D printing. Gage: Testing PCBs Ian: Integration, prep for demo. Wrote motor configuration and device driver. Refactored control script.
Apr 12 – Apr 18	Carson: Worked on 3D modelling and prepped for mock demo. Gage: Polish or create work arounds for any final issues. Ian: Wrote hammer control protocol. Bug fixes, worked on 3D modelling. Prep for demo/presentation
Apr 19 – Apr 25 **Mock Demo/Presentation**	Carson: Final bug fixes, prep for demo/presentation. Worked on 3D modelling Gage: Final bug fixes, prep for demo/presentation. Worked on 3D modelling and printing components. Ian: Final bug fixes, prep for demo/presentation. Worked on 3D modelling
Apr 26 – May 2 **Final Demo/Presentation**	Carson: Prepped for demo/presentation Gage: Prep for demo/presentation Ian: Prep for demo/presentation
May 3 – May 9 **Final Papers Due – May 6** **Lab Notebook Due**	Carson: Cleanup and submission (final project, bot competition, lab notebook). Gage: Cleanup and submission Ian: Cleanup and submission

5. Conclusion

Overall, we were successful in what we set out to do in this project. We constructed our ant-weight battle bot and competed in the end-of-year battle bot competition, and although we didn't win, our bot was fully functional and had all the features we had proposed in the beginning of the semester. Our bot was resilient to inversions via an accelerometer, could automatically detect a rival bot with an accelerometer, and could deliver a blow via the hammer arm. Our PCB was fully functional, our software was bug-free, and our chassis was robust. All our components were custom-made, and we feel very proud of the work that we've done. There were, however, a small number of things we weren't satisfied with, and if given the chance, would improve upon in the future. First off, as mentioned in section 3, the accelerometer chip we used took over a second to raise the inverted interrupt flag. We would replace this chip with a more performant one so we could detect flips faster. Secondly, our hammer was not as 'crushing' as we would have hoped. We would replace the high-torque motor we had chosen with one more suited for this application, specifically one that could generate much higher RPMs. Finally, during competition one of our wheels was knocked off, which ultimately cost us the match. We would improve upon our chassis by creating thicker wheels with real treads, and placing them inside the housing instead of being external to the bot. Given those changes, we are confident our bot would fare much better in the competition. We hope that our project overall serves as an inspiration for others to get into robotics, but for the benefit of society, such as search-and-rescue or agriculture drones.

One important thing to mention in this context is the ethical responsibility surrounding a battle-bot meant for destroying another bot. We want to emphasize that while our bot was designed to do damage, it was designed and used strictly to compete in the end-of-year battle-bot competition, which took place in an enclosed and supervised environment. We never did, and never will, use our bot to harm people, animals, or property. The IEEE Code of Ethics I.1 [3] states that we as engineers are “to hold paramount the safety, health, and welfare of the public, to strive to comply with ethical design and sustainable development practices, to protect the privacy of others, and to disclose promptly factors that might endanger the public or the environment”, and we commit ourselves to upholding that as much as we can. During development, we also strove to uphold the other values of the code, such as treating all people fairly and with respect, and upholding each other to the same values.

References

[1] Adafruit Industries, “Ultrasonic Distance Sensor – 3V or 5V – HC-SR04 compatible – RCWL-1601,” 2019. Available: https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/307/4007_Web.pdf. (Accessed: Feb. 9, 2026).

[2] Espressif Systems, “Bluetooth® Main API,” *ESP-IDF Programming Guide*. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/bluetooth/esp_bt_main.html#api-reference. (Accessed: Feb. 9, 2026).

[3] IEEE, “IEEE Code of Ethics,” 2020. Available: https://www.ieee.org/about/corporate/governance/p7-8. (Accessed: Feb. 12, 2026).

[4] Memsic Semiconductor Co., Ltd., “MC3416 3-Axis Accelerometer,” 2021. Available: https://www.digikey.com/en/products/detail/memsic-inc/MC3416/15292804. (Accessed: Feb. 9, 2026).

[5] “Motor Control Pulse Width Modulator (MCWPM),” Espressif. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/mcpwm.html. (Accessed Apr. 8, 2026).

[6] “How gap and GATT work - bluetooth low energy basics,” Punch Through. Available: https://punchthrough.com/how-gap-and-gatt-work/ (Accessed Feb. 25, 2026).

[7] “Nimble-based host apis,” Espressif. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/bluetooth/nimble/index.html (Accessed Feb. 25, 2026).