Autonomous Golf Cart Firmware

Gerik Kubiak

Autonomous Golf Cart Firmware

Table of Contents

Abstract 3
Project Overview 4
Golf Cart Hardware 4
Golf Cart Sensors 6
Main Computer 8
Control Board 8
Zeus – Board Power Supply 10
Janus – Forward Neutral Reverse Control 10
Hermes – Throttle Control 12
Iris – USB-USART Bridge and 7 Segment Display 13
Hera – Sensor Connectors 13
Hephaestus – Steering Motor Drive 13
Apollo – Lights 14
Dionysus – Battery Voltage Monitoring 14
Firmware 14
USART 14
I2C 15
ADC 16
USB 17
Timer Callbacks 17
Firmware Drivers 18
Steering Driver 18
Speed Control 18
Communication 19
USART Command Line 19
USB Communication Interface 20
Future Improvements 20
Conclusion 21
Appendix A – Control Board Schematics 22

Abstract

The Autonomous Golf Cart Project is a project sponsored by the Cal Poly Robotics Club. The multidisciplinary team is adding sensors and electronics to a regular golf cart with the goal to drive the golf cart around campus without and human input. This task requires a plethora of hardware and firmware to control that hardware. The firmware provides an interface for higher level software to then control the hardware and therefore drive the golf cart. This report is focused on the hardware modifications and the firmware used in order to drive the golf cart from a computer.

Project Overview

The Autonomous Golf Cart Project is a project within the Cal Poly Robotics Club to build a golf cart which can autonomously drive around Cal Poly's campus. The team for the project consists of around 15-20 members from the Computer, Electrical and Mechanical Engineering majors. The project started in spring of 2013 and has gone through many phases and revisions. The current revision of the golf cart started in fall of 2015 and started with a redesign of the control board, a printed circuit board (PCB) which controls the electronics of the golf cart. At the heart of this board is a microcontroller, which runs the firmware that acts as a bridge between the high level autonomous algorithms and the low level electronic controls. This senior project was tasked with writing the firmware necessary to control the golf cart from a computer. An image of the golf cart is shown in figure 1.

Figure 1: The Golf Cart

Golf Cart Hardware

The golf cart has been retrofitted with electronics to enable the entire system to be controlled from a computer. The golf cart uses a 2 Horsepower electric motor connected to the rear axle in order to drive. This motor is powered off three 12 V batteries in series to create an effective 36 V power source. This supply is connected through a power solenoid, to a speed controller, and then to an H-Bridge.

The power solenoid is a safety feature to ensure that the motor only receives power when someone is on the golf cart. The power solenoid is activated by arming an emergency stop switch, and by slightly depressing the pedal. When both these events have occurred, the solenoid connects and allows current to flow into the H-Bridge and speed controller.

The drive motor H-Bridge is made up of three solenoids which together control the direction of current through the drive motor, thereby controlling the direction of travel. The H-bridge is controlled with two wires, forward and reverse. By pulling the forward wire to ground the H-bridge will switch into a forward configuration. If the reverse wire is pulled to ground, then the H-bridge will switch into a reverse configuration. Leaving both pins floating will leave the golf cart in neutral and the motor disconnected. Pulling both pins to ground is an illegal state and must be avoided. If this occurs the 36 V battery will be shorted to ground. The batteries are protected by a fuse which will trip if this state occurs. During manual operations the H-Bridge is controlled by a physical switch. The wiring of this switch prevents the invalid state from ever occurring, but this is not guaranteed when in autonomous mode. Autonomous mode has other safety features to prevent this state from occurring. When in this mode, the H-Bridge is controlled by the Janus board. Because this H-Bridge controls the forward, neutral, and reverse state of the golf cart it is referred to as the FNR H-Bridge.

The speed controller in the golf cart is controlled through a voltage applied to one of its pins. A higher voltage on this pin will result in a higher overall speed of the golf cart. During manual operation this pin is connected to the wiper of a potentiometer controlled by the pedal. By pressing the pedal down the wiper is moved further up the resistor, resulting in a larger voltage. In order to control the golf cart autonomously, the potentiometer is wired through the Hermes board, which either passes the signal through or sets its own voltage.

The steering system of the golf cart consists of two steering shafts connected in parallel. One of the shafts is connected to a traditional steering wheel, allowing the golf cart to be steered manually. The other shaft is connected to a DC motor. By engaging and applying power to the motor the golf cart can be steering either left or right. This DC motor is controlled by a second H-Bridge located on the Hephaestus board. In order to get the current steering angle, a linear potentiometer is located near the steering wheels. This device acts like a normal potentiometer, but the wiper is connected to a string. As the string is pulled the wiper moves and the voltage on the wiper node changes. As the golf cart steers, the linear potentiometer string is pulled. This creates a voltage which directly correlates to a steering angle. This information is fed into the main MCU in order to properly steering the golf cart autonomously. An image of the golf cart internals is shown in figure 2.

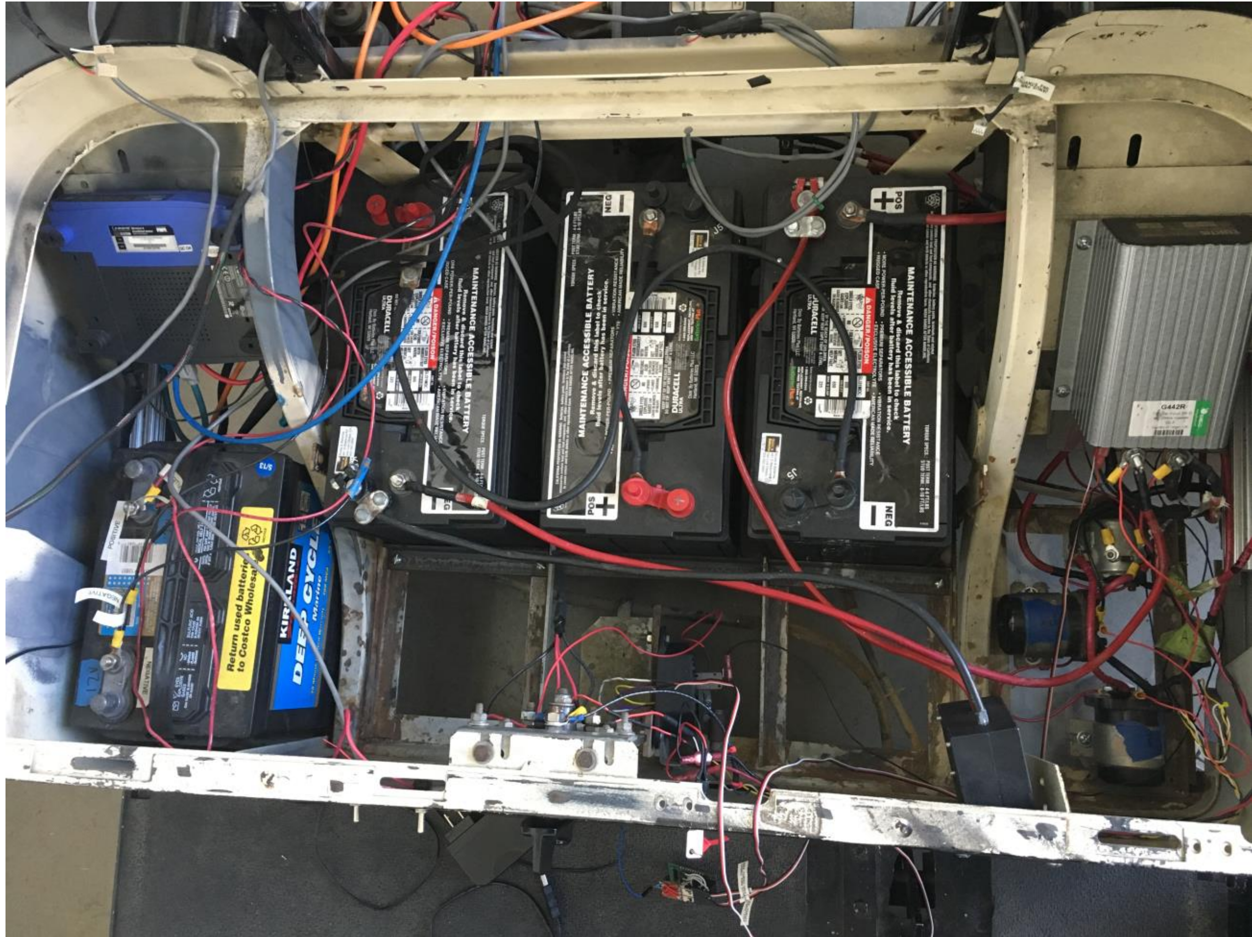

Figure 2: The Internals of the Golf Cart. The speed controller and the drive H-Bridge are shown on the right

Golf Cart Sensors

The golf cart uses multiple sensors to gain information about its surroundings. First and foremost is the LIDAR, which is similar to a radar, but uses light instead of sound waves. By bouncing light pulses off of surfaces and recording the time-of-flight for each pulse, the distance to the surface can be measured with a high degree of accuracy. The LIDAR used on the golf cart scans 270° with 0.5° resolution at 30 Hz. This is known as a 2D LIDAR as it scans its surroundings on a 2D plane. Sample LIDAR data, along with a picture of the scene is shown in figures 3 and 4.

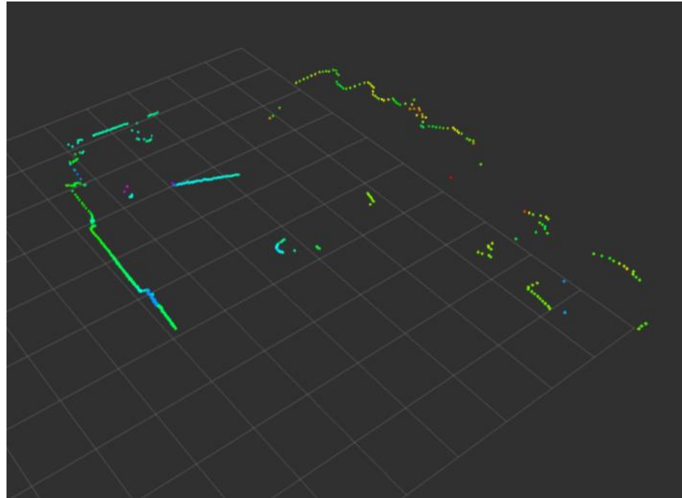

Figure 3: Sample Lidar Data. Each point represents a received light pulse. The horizontal bar in the top left is the whiteboard in figure 4

Figure 4: Sample Lidar Data Scene

Although the LIDAR is very useful at detecting objects, it can be difficult to determine what an object is. A camera on the other hand is useful for determining what an object may be, but it is significantly more difficult to detect objects, especially comparted to using a LIDAR. Therefore, a camera will be used to aid in object recognition. The camera is not currently being used, but it is planned for future use.

Because of both the viewing angle and position of the LIDAR, the LIDAR cannot see behind the golf cart. Also, because the LIDAR can only see on a 2D plane, objects below the LIDAR are invisible. To help with object detection in these cases, ultrasonic sensor will be used. These sensors use the time of flight of sound to detect an object and are significantly cheaper than a LIDAR. These sensors will go on the sides and back of the golf cart to aid in object detection.

The golf cart also contains two hall effect sensors on the front wheels. A circle of magnets is also present on the inside of the wheels. When the sensor passes over a magnet it toggles its output, either from low to high or high to low. Therefore, when the golf cart is at a steady speed, the hall effect sensor outputs effectively a square wave. The frequency of this square wave is related to the speed of the golf cart and higher frequencies indicate faster speeds. Counting the number of transitions from high to low and low to high also provides information of the total distance travelled.

A GPS and Inertial Measurement Unit (IMU) are also used to aid the hall effect sensor in detecting position. The GPS returns raw longitude and latitude coordinates, but is only accurate to about +/- 5 meters or so. the IMU contains and accelerometer and gyroscope internally, which can be used to determine the velocity and angle of the golf cart.

Main Computer

The microprocessor on the control board is not powerful enough to read in the LIDAR and camera sensors. For this reason, a computer running Linux is located on the golf cart. The computer run Robot Operating System (ROS), which is a popular framework for passing messages between processes. ROS is used many places in industry and therefore many help processes, known as nodes, exist to help with filtering, formatting and combining raw sensor data. The main computer uses the ROS framework to make a decision about where to move the golf cart, and then sends a message down to the control board using USB. The control board will then control the golf cart's electronics to move the golf cart.

Control Board

The control board is not a single PCB, but a collection of smaller PCBs. All these smaller boards, or daughter cards, connect to single mother board known as Olympus. This approach allowed the controls boards to be developed in parallel. It also simplifies each board, reducing the chance for mistakes. Also if a single board breaks, only that board needs to be fixed. The next sections detail each of these control boards. An images of the control board with and without the daughters are shown in figures 5 and 6.

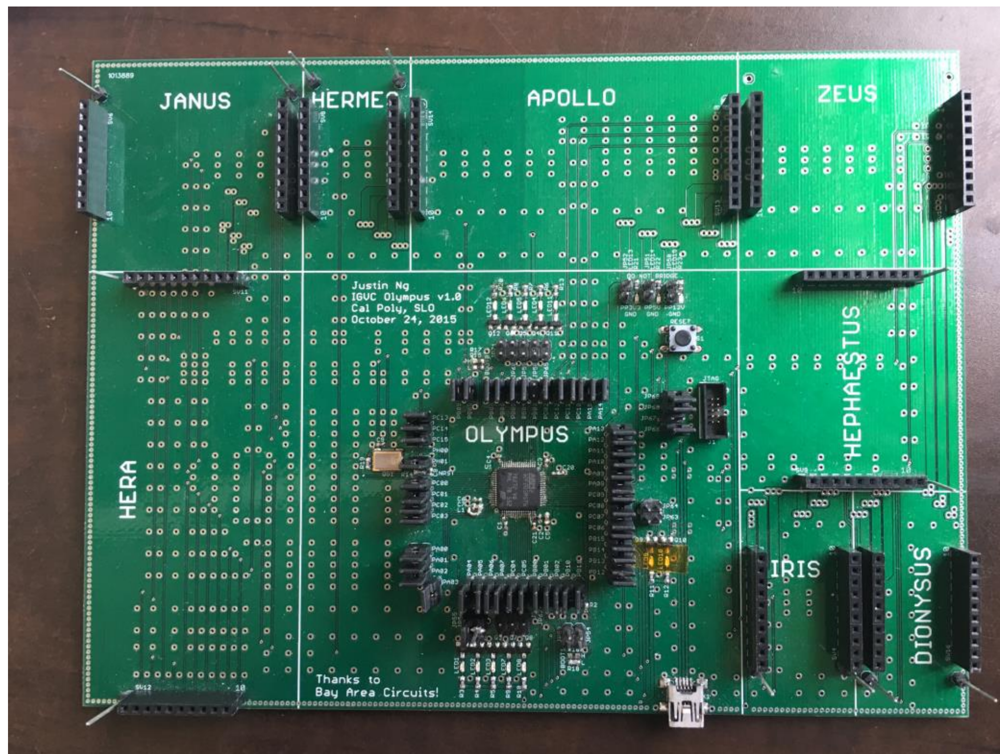

Figure 5: The Control Board without any Daughter Cards

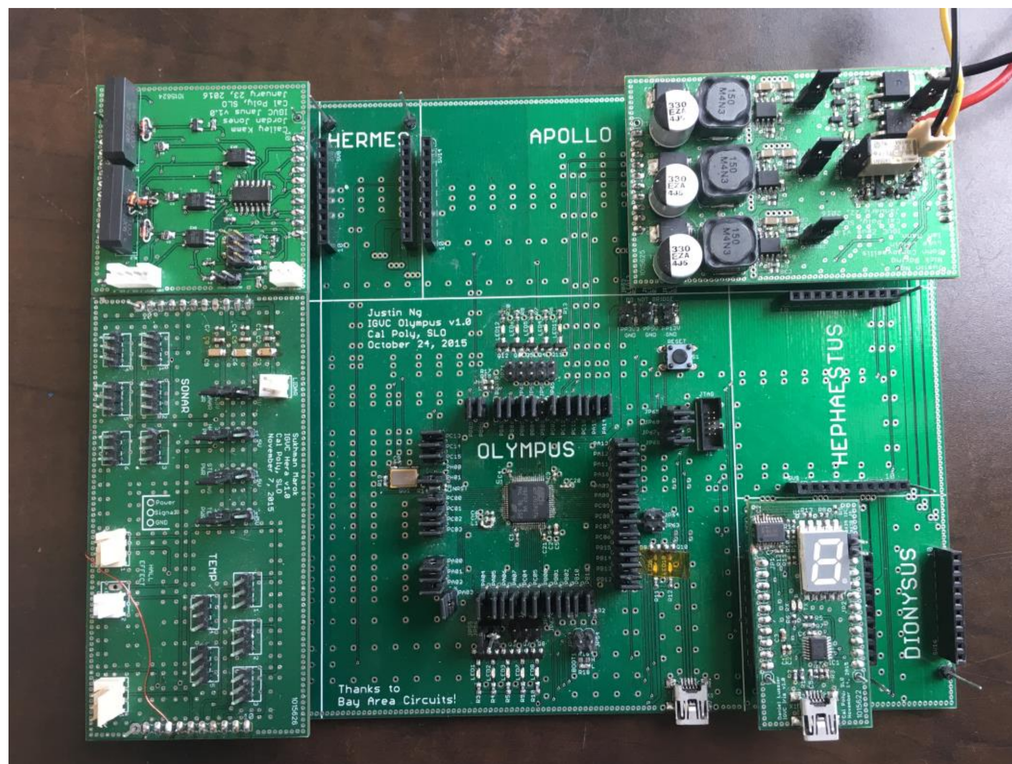

Figure 6: The Control Board with a few Daughter Cards Installed

Olympus - Motherboard

Olympus houses the main microcontroller (MCU), an STM32F205 Cortex-M3 ARM microcontroller. It also integrates some debug LEDs, a USB port directly connected to the pins on the main MCU, and a JTAG port to program the main MCU. The Olympus board also acts as a motherboard, with slots for 7 daughter cards. The Olympus board does not contain any way to directly interact with the golf cart. This task is instead left to the daughter cards. The schematics for the Olympus board are shown in Appendix A.

The main MCU is also not directly connected to any of the daughter card ports. Each pin is connected through a jumper and then to the daughter card pin. By doing this and pin can be disconnected from the microcontroller and the entire pinout can be reconfigured after the board has been produced. The downside to this technique is that it takes up significantly more board space and the signal integrity is weakened. Fortunately, the size of the daughter cards leaves a large amount of space for the relatively small Olympus circuit. Also, the fastest signal used is full speed USB, which has a maximum bandwidth of 12 Mbps. It is not ideal to run this signal through a jumper, but it seems to work well enough.

The USB port on Olympus is connected directly to the pins on the main MCU and is used for the communication protocol used between the Linux box and the control board. All the ROS commands to control the golf cart get sent through this port.

Zeus – Board Power Supply

Zeus regulates a 4S, 14.8 V Li-Po battery down to 12, 5, and 3.3 V rails used by various circuits on the board and golf cart. These voltage levels are created through the use of step-down, buck converters. The only things on the golf cart not powered by the Zeus board are the two motors and the Linux computer. The board also provides reverse and over voltage protection for the input voltage and integrates a 12-bit ADC to read the input and rail voltage as well as the input and rail currents through the use of shunt resistors. The schematics for Zeus are in appendix A.

The ADC voltage and current values are constantly read by the MCU. These values are then converted into a decimal voltage value through multiplication and division by a predefined constant. Although this constant could be calculated from the resistor and reference values in the schematic, it is more accurate to take a measurement and correlate the measured values to the ADC values than to use the calculated values. This allows cheaper, less precise resistors to be used, but the scaling factors produced could not be used on another board as easily.

Janus – Forward Neutral Reverse Control

The Janus board controls the solenoid H-Bridge in the golf cart, allowing the golf cart to travel both forward and reverse. This is done by pulling either the forward or reverse wire to ground through the use of a relay. The board also contains a small microcontroller which prevents both the forward and reverse wire from being pulled to ground. The use of a microcontroller, instead of discrete logic, allows for more flexibility and allows for a delay to be added into the switching logic.

The Janus board is responsible for interfacing between the main MCU and the mechanical H-bridge inside the golf card. The board allows the main MCU to both set the FNR H-bridge and detect its position. Setting the FNR state is done with two relays. When current is allowed the flow through the

relay's coil, the associated wire, F or R, is connected to ground. These relays are controlled by an N-FET, which is connected directly to the output for an ATTiny 84 microcontroller. This microcontroller is responsible for controlling these N-FETs, and therefore the relays. It is also responsible for never asserting both N-FETs at the same time as that would cause the FNR H-bridge to enter the illegal state.

Janus is able to detect the position of the FNR H-bridge by checking if the Forward and Reverse wires have been pulled to ground. This is done through two optocouplers which electrically isolate the board from the batteries. The state of these pins is also fed into the ATTiny MCU to send to the main MCU. The schematics for the Janus board are provided in appendix A.

The main MCU sets the state of the FNR H-bridge using two pins, an enable pin and a direction pin. The pins follow the truth table shown in Table 1.

Table 1: Main MCU FNR Control Truth Table

Enable Pin State	Direction Pin State	Golf Cart FNR state
0	0	Neutral
0	1	Neutral
1	0	Forward
1	1	Reverse

The ATTiny MCU uses this truth table to control the FNR H-bridge. The use of an enable and direction pin, as opposed to a forward and reverse pins means that the main MCU can never try and send an invalid state to the ATTiny MCU, which reduces the chance of an illegal state.

The main MCU is able to read the state of the FNR using I2C from the ATTiny. The ATTiny acts as an I2C slave and responds with a single byte representing the state of the FNR H-bridge. The main MCU polls the ATTiny on occasion in order to read the state of the FNR H-bridge. This information is not much use to the main MCU, but it is useful as a debugging tool and as feedback for the end user.

The ATTiny is a very small 8-bit microcontroller, containing 8 kilobytes of flash space and 512 bytes of ram. The ATTiny firmware is composed of two parts. The first part uses the direction and enable pins for the main MCU to control the FNR H-bridge. This code constantly checks the state of the direction and enable pins to see if they have changed. If the MCU has gone from either forward or reverse into neutral, the ATTiny stops pulling the forward or reverse wires and the golf card immediately enters the neutral state. If the MCU has switched into forward or reverse though more care must be taken. It takes a non-zero amount of time to switch the solenoids that make of the FNR H-bridge. This is accounted for in the ATTiny MCU by forcing a delay between changing state. When switching into forward, the ATTiny first switches to neutral for 250 ms, allowing the solenoids to fully switch into a neutral state. After this delay the ATTiny then sets the FNR H-bridge into forward mode. If there was no delay, the golf cart could briefly end up in an illegal state while switching FNR states and briefly short the battery. This same sequence occurs when switching into reverse. This code is compact and has been thoroughly tested to ensure that the illegal state never occurs.

The second half of the firmware is responsible for responding to I2C commands. The ATTiny does not contain an I2C peripheral, but instead contains a Universal Serial Interface (USI). The USI peripheral is composed of a 4-bit counter, a shift register, and a couple multiplexers. With a bit of configuration this

peripheral can act as a I2C slave. The USI has an I2C start bit detector, which can cause an interrupt. The counter ticks on both external clock edges and can cause and interrupt on overflow. By combining these two interrupts a fully functional I2C slave can be configured. On the start bit interrupt the state of the ATTiny is reset, causing it to wait for its slave address. Then, when the counter overflows, the ATTiny checks the data in shift register to see if it matches its slave address. If there is a match, then the shift register is loaded with the state of the FNR H-bridge and the address is acknowledged. During the next byte the USI peripheral will automatically output the state of the FNR H-bridge over I2C and the I2C transaction will be complete. If the address is not matched, then the ATTiny will stop the SDA line from outputting data and the other I2C slaves on the bus are free to drive the data line.

The ATTiny has been working very well for multiple months and although the USI peripheral was awkward to work with at first, it has been running very well. Using the ATTiny to has also made development on the main MCU less stressful, as no invalid states on the main MCU can cause an invalid state for the FNR H-bridge.

Hermes – Throttle Control

The throttle of the golf cart is controlled through a speed controller which sits between batteries and the motor. The speed controller is a bit of a black box, but provides a simple interface in order to control the speed of the golf cart. By applying a voltage between 0-13 V, the speed of the golf cart can be controlled. To control the speed electrically though, a 6-bit digital to analog converter (DAC) is used. The DAC generates a 0-3.3 V signal which is later amplified by an op-amp to recreate the 0-13 V signal that is sent to the speed controller. The Hermes board integrates the DAC and op-amp along with circuitry to switch between manual and automatic speed control. The board also acts intermediary between the pedal potentiometer and the speed controller. Hermes automatically passes this voltage through to the speed controller, even without power present. The schematic for the Hermes board is shown in appendix A.

Unfortunately, a few flaws were found in the schematic of Hermes after the board was fabricated. Namely, the board required power to pass through the pedal potentiometer voltage. To fix this most of the switching circuitry was replaced with a relay. This relay allows the potentiometer voltage to pass through without power and then physically switches to use the DAC voltage when a pin is asserted by the main MCU.

The Hermes board is controlled by an enable pin and an I2C interface. The enable pin controls the state of the relay and therefore switches between manual and automatic speed control. By setting this pin high through a GPIO the automatic steering mode is enabled and the DAC controls the speed of the golf cart. The DAC is controlled by sending a single byte over I2C. The DAC then converts the higher 6 bits into a voltage from 0-3.3 V.

Although the control interface is simple, the main MCU has a few safety features built in to prevent damaging the speed controller. Firstly, when a request to change to a fast speed is received, the main MCU does not instantaneously set the higher speed. Instead the main MCU ramps up the speed. This is in part to prevent jerking the golf cart and also a reliability feature. The speed controller has failed before when a voltage was instantly applied to its input terminal. This resulted in smoking and completely broken speed controller. The exact reason for the failure could not be diagnosed so to

prevent further problems the speed is controlled in a way that would model human operation. The firmware also ensures that the golf cart is in either forward or reverse before ramping the voltage and that pedal pressed is before the main MCU will start to ramp the voltage. If this is not the case, the main MCU will default to 0 V, which stops the golf cart. This technique complicates the control code, but helps to protect the golf cart.

Iris – USB-USART Bridge and 7 Segment Display

The Iris board does not contain an interface to the golf cart like the Janus or Hermes board, but rather provides useful debug output for writing firmware. The first main function of Iris is its USB-USART bridge. This chip (an FT230XS) converts USART serial signals from the main MCU to a USB-CDC interface which can be connected to a computer. The USART peripheral is one of the easiest peripherals to get working and can provides simple debug status messages during the initial bring up. The Other half of the Iris board contains an I2C GPIO expander and a 7 segment display. The display provides a way to communicate the status of the golf cart without having to be connected to the USB interface. As of now, the display shows the current FNR state, which is useful for ensuring the H-Bridge is engaged before trying to drive the golf cart.

The USB on Iris is not only for bring up though. It also provides a basic terminal that can be used to probe the status of the golf cart and issue commands. This command protocol is detailed in the USART Command Line section.

Even though the board is simple, Iris is one of the most used boards as it acts as an interface between the user and the main MCU. The schematics for Iris can be found in Appendix A.

Hera – Sensor Connectors

The Hera board contains connectors which allow the board to interface with external sensors. The board contains inputs for two hall effect sensors, five temperature sensors, six ultrasonic sensors, and a steering potentiometer. The board also contains a connector to power the LIDAR. The hall effect sensors and ultrasonic sensors are all connected to separate timers on the main MCU, which allows the MCU to efficiently determine the pulse width of digital data sent from the hall effect and ultrasonic sensors. The temperature and steering encoder both output analog voltages which are connected to the ADC of the main MCU. The schematic for Hera can be found in Appendix A.

Hephaestus – Steering Motor Drive

Hephaestus contains a solid state H-Bridge which is used to control the steering motor on the golf cart. The board also contains a current sense circuit so that the main MCU can log the current consumption of the steering motor. The Steering Motor is powered through an isolated 12 V battery and also contains circuitry to isolate the board from the main MCU through optocouplers. The schematics for the Hephaestus board are still being finalized.

The main MCU communicates to the steering motor H-Bridge through the use of two pins, a direction and a PWM pin. The direction pin controls the direction that the golf cart steers in, while the PWM pin

acts as a speed control and enable pin. If the PWM pin is held low, then the steering motor receives no power. When the pin goes high, the steering motor receives 12 V across its inputs. By turning the PWM pin on for only a fraction of the time, the total power to the motor can be decreased and the speed of steering can be more finely controlled.

Apollo – Lights

The Apollo board is responsible for controlling the rear and flashing lights on the golf cart. In order to reduce the pinout of the board, the board uses a shift register to control the lights. This allows the board to control 16 lights using 3 pins from the main MCU. The schematics for the Apollo board are still being finalized.

Dionysus – Battery Voltage Monitoring

The Dionysus board contains multiple ADCs in order to measure the voltage of the four lead-acid batteries inside the golf cart. This information can then be used to roughly estimate the remaining capacity of the golf cart's batteries. Dionysus has not been built yet and is still in the design phase.

Firmware

Much of the firmware involves reading and writing the MCU's peripherals. The firmware for the control board is built to reduce the opportunity for race conditions. This is done through the use of service functions called from a main loop. While Interrupt Service Routines (ISRs) are used throughout the code, work actually done in their routines is minimal. This work is limited to copying data out of a peripheral and setting a flag to indicate the further processing needs to be done. In the main loop, these flags are constantly checked inside service functions to in order to process the peripheral. While the actual work done in the ISR vs. service function is peripheral dependent, the service function usually deals with starting separate transactions. The details of each peripheral are explained below.

USART

The Universal Synchronous Asynchronous Input Output (USART) peripheral is used to provide a console interface for the control board. At a hardware level, the USART consists of two pins, Rx and Tx. These pins are each half-duplex and are used to send data out (Tx) or read data into (Rx) the MCU. Data is sent one byte at a time without a clock signal. This requires both devices on the bus to agree on a clock speed for transmission. USART is used because it is easy to get up and running and is well suited for simple debug interfaces.

The USART driver makes use of the Direct Memory Access (DMA) peripheral in order to write data to the USART efficiently. When using the DMA, the firmware specifies the start location and length of the data to send over the USART. By setting a few more configuration registers, the data is automatically transferred out of the USART peripheral without needing any MCU intervention. This frees up the MCU

to do other tasks while USART data is being transferred out. When the DMA is done transferring data, it fires an interrupt to signal that it has completed.

The driver maintains a list of multiple output buffers. Data to be written out the USART peripheral is first copied into these buffers. Once a byte has been copied into these buffers, if no USART DMA transfer is currently occurring, all the data in a single output buffer is copied out using the DMA. If more data arrives before DMA transfer has finished it gets placed into the next available output buffer. Then, when the current DMA transfer finishes, it sets a flag, informing the firmware that the USART Tx peripheral needs to be serviced. When the USART's service function is later called in the main loop, it checks to see if more data is waiting to be transferred. If so it will start another DMA transfer, otherwise it will wait for more data.

Although it would be possible to check for more data in the DMA ISR, it might introduce difficult to debug race conditions if more data was being written to the USART peripheral as a transfer finished. Although this would need to eliminate the need for a service function as well as increase the communication bandwidth, it would complicate the code to copy data into the USART input buffers. Restarting transfers in the service routine also decreases the time spent in the ISR, which is generally a good thing.

The USART driver makes use of a second DMA to read in data over the Rx line. This DMA is configured in a similar way to the Tx DMA, but instead of copying data from memory to the USART peripheral, it copies bytes from the USART peripheral to memory. The Rx driver only uses a single buffer, unlike the Tx driver. The DMA is configured with the location and length of this buffer, but is also configured in circular mode. This means that when the DMA has finished writing the entire buffer it automatically begins writing data at the beginning of the buffer again. This allows the Rx DMA to operate constantly without MCU intervention. The only time the MCU is involved is when it wants to read a byte from the input buffer. The MCU keeps track of the next location of valid data. When another part of the firmware requests to read a byte, the driver checks to see if the current location has valid data, and returns the byte if it does.

Through the use of these two drivers, the USART peripheral is efficiently able to copy data in and out of the peripheral. There are still improvements that could be made, however. Currently, the Tx driver will only copy a single buffer out to the peripheral at once, even if multiple buffers are full. These buffers are located in consecutive memory locations and therefore it should be possible to set the DMA to transfer multiple buffers at once. This would not be too difficult of a change, but it is not pressing enough to fix right now.

I2C

The Inter Integrated Circuit (I2C) peripheral is used to talk to multiple sensors on the golf cart including the Janus FNR ATTiny (for current FNR state), and the Zeus ADC (for reading the voltage and currents of all the power rails). It also used for writing to the Hermes DAC (for controlling the golf cart speed), and the Iris seven segment display (for displaying information on the board). I2C uses only two pins, a clock pin (known as SCL) and a data pin (known as SDA), to communicate to all these devices. Each of the devices is considered an I2C slave, and is assigned an address. When a slave detects its address on the SCL and SDA lines, it responds with an acknowledgement. From there, the MCU, known as the I2C

master, can choose to either read or write data to the slave device. After the I2C master has finished talking to a slave, it sends a stop bit over the lines to signal the end of communication. From there the master is free to talk to another slave.

The I2C driver works by storing a queue of I2C transfers. When another part of the firmware wants to either send or receive data over I2C, it creates an entry in the queue. The I2C driver processes each element in the queue in the order they are received (FIFO). A queue element contains information about the transfer, including: the slave address, the number of bytes to transfer, whether the transfer is writing to or reading from the slave, and a callback to call once the I2C transfer has completed.

When a write transfer is setup, the I2C driver stores the bytes to write into a small FIFO. This FIFO is shared between all the I2C transfers, but because each transfer is processed in the order it was received, each transfer will always pop out its own bytes. When a new byte needs to be sent, an ISR is generated. In this ISR, a byte is popped from the FIFO until all the bytes have been transferred. Once this occurs, the ISR sets a flag signaling that the transfer has completed. This is then processed by the I2C service function in the main loop. From here, the callback for the transfer is called and the next transfer is started if available.

Reading data over I2C is similar to writing, but with a few differences. Instead of writing data to a FIFO, an array is used to store the data received from the I2C peripheral. Once the transfer is completed, the callback is called, but includes the location of this array as an argument. The callback is then free to copy this data to its own buffers.

This I2C driver is different from the USART driver in that it actually does some work in the ISR. The service function is still called between individual transfers, while the ISR has code to continue the I2C transaction until it is completed. This approach was used to shorten the total I2C transfer time. To send two bytes over I2C requires four ISR calls. If each of these calls had to wait until the main loop had time to service them, the total transfer time could become very long.

ADC

The ADC is currently used to read in two devices: the steering wheel angle and the accelerator pedal position. There are also headers on the Hera board for four other temperature sensors, which can be used if more analog inputs are needed in the future. The ADC makes use of the circular mode of both the ADC and the DMA in order to run continuously with no processing time required.

Every pin that can be used as an analog input is assigned a channel by the hardware. This channel is then used by the ADC to select a pin to read and convert. The ADC can be given a list of channels to convert. As soon as the first channel in the list is converted, the ADC automatically converts the next channel. When using a polling or interrupt based approach, care must be taken to read this value quickly or else it will be overwritten by the next conversion. This is less of a problem when using the DMA though. By setting the DMA priority to its max value, it would be very unlikely that data would not be transferred out of the ADC. Along with a list of channels, the ADC can also be set into circular mode. This means that once all the channels in the list have been converted, the list begins again. This continues to happen forever, keeping the ADC running continuously. By setting up the DMA in circular mode as well, the ADC

can be continuously read by the DMA into a buffer. From C, this looks like an array that is constantly being updated with the latest ADC readings.

This method can be improved further. By setting the DMA buffer size to eight time the necessary size (number of ADC channels * 2 bytes per channel * 8), the DMA will store the last eight readings automatically. Then, in software, the MCU can add up all readings and shift the sum by three, effectively applying a low pass filter over the ADC readings. Overall, this approach in very efficient as it requires zero CPU cycles in order the gather the ADC data, and only a few cycles to apply the low pass filter over the data.

USB

The main MCU supports native low and full speed USB (USB 1.1) without the use of an external PHY. The control board implements a USB-CDC interface in order to receive commands from the Linux computer. The USB peripheral, as well as ST's USB stack, provide most of the functionality in order to implement this interface. A few callbacks are registered with the USB stack in order to interact with the stack. The first callback is called on USB initialization and is used to initialize various buffers for use in the stack. The second callback is for deinitialization and is currently unused. The third callback is for CDC control messages and is also unused. The final callback is called whenever data is received in a USB OUT packet. This callback stores all the received bytes into a buffer that is later read from the main loop.

The USB stack also provides a system for sending data out the USB port. A transfer is started by passing the location and size of the buffer to send to the USB stack. This data is not copied immediately though, but is copied straight out of memory on a later USB interrupt. This means that the data must be placed into a buffer until it is actually copied out of memory. This caused an issue in an earlier version of the code where the buffer to send was stored on the stack. When the USB interrupt finally occurred the buffer had been changed, causing weird data to be sent to the Linux computer.

The current transmit implementation is almost identical to the USART driver, where a set of buffers is used to store future transactions. Each time a buffer is passed to the USB stack, it is marked to prevent further modifications. Once the USB data has been actually transmitted in a USB IN packet, the buffer can then be modified for other USB transactions.

Timer Callbacks

Although not directly a peripheral, the firmware uses one of the timer peripherals in order to schedule periodic tasks. The timer is set up to cause an interrupt every millisecond. Each task registers a callback function, a void pointer which is passed to the callback function, and a period in milliseconds. The firmware uses the period to store the number of milliseconds until the callback should be fired. Each time the timer interrupt occurs these values are decremented. When one reaches zero, the associated callback is called. The callback then does its work and returns one of two values: DISABLE_TIMER or CONTINUE_TIMER. DISABLE_TIMER disables the callback and removes it from the utility. The CONTINUE_TIMER resets the time until the callback to the period provided on initialization allowing it to fire again.

This timer callback mechanism is used in multiple areas of the golf cart. It is used to schedule many of the I2C transfer as well as control the steering on the golf cart.

Firmware Drivers

It is not enough to just read from the sensors. In many cases it is necessary to react to sensor readings directly in the firmware.

Steering Driver

In order the steer the golf cart, the firmware must read from the steering potentiometer to detect the current angle of the steering wheels, and then write to the steering H-Bridge on the Janus board in order to actually steer the front wheels left or right. The steering driver makes use a callback timer, and is currently called every 100 milliseconds, or at 10 Hz, although not much work has been done to find the optimal value.

In order to enable steering, the Linux box provides the control board with an angle to steer to, in the form of a number from 0-65535. This is then mapped to a target ADC value to reach. This target ADC value represents the final angle of the golf cart. On every call of the steering callback, the firmware checks that steering has been enabled and if so steers the golf cart left or right until it is in the correct location. As of right now the steering motor is either full on in a direction or off. This makes it difficult to steer to an angle precisely because there is no control of the speed of turning. In order to try and prevent the steering for oscillating around the correct angle, the steering code has a dead zone and therefore accepts a steering value within some range of the correct value as being correct. This means that steering to a single angle is less precise than it could be, but it allows the golf cart to eventually finish steering, instead of constantly overshooting. Once the driver has noticed that the golf cart has steered to correct value it then disables steering until a new steering target is provided by the Linux computer.

This control loop could also have been implemented in Linux. By passing the current steering angles back to the computer, the computer could then control the direction of the steering motor and steer the wheels to the correct angle. This was not done for two reason. First, by writing the driver in firmware the total control loop is tighter. The control can react much to changes in steering angle much quicker than the computer could which results in less oscillation in reaching the final angle. Also, the current method abstracts the implementation of steering from the computer. If the control board changes in the future, no change is required on the computer side, just the firmware needs to be rewritten.

Speed Control

The speed of the golf cart is controlled through the use of a 6 bit DAC on the Hermes board. A DAC value of 0 relates to a speed of 0 mph, while a DAC 63 relates to top speed, around 10 mph. Although it would be simple enough to just set the DAC to a value to select our speed, a more complicated approach is used to protect the golf cart and provide a better user experience. In initial tests with the Hermes board, when the speed was set fairly high, and the pedal was pressed, the speed controller made a popping

sound and started smoking. As the speed controller in the golf cart is mostly a black box, it is hard to know exactly what went wrong. To prevent this from happening with future speed controllers, the firmware tries to emulate how a human driver would select the speed.

When the golf cart is stopped and the firmware receives a command to set the speed to a certain value, it store that value as the target speed value. Then it does two checks. The first check is to make sure the golf cart has been put into either forward or reverse mode. If it is currently in neutral, the speed will automatically be set to zero. The second check is that the pedal has been pressed. Once the pedal has been pressed passed a certain point, the pedal check will pass. While there are hardware safety features to prevent the golf cart from driving when the pedal is not pressed, putting the check in firmware further makes the automatic control look as though a human were driving the golf cart. Once these two checks pass, the firmware begins ramping the speed. Every 200 ms the DAC value will increase by 5 until it reaches the target speed value. This makes it look as though a human were slowly pressing the pedal, as opposed to a sudden step in speed. The ramping on speed also allows the golf cart to accelerate more slowly and helps to prevent sudden jerks.

There are two methods used to control the speed. In the first method, the firmware simply receives a value that should be set as the DAC value. The firmware will ramp to this DAC value and stop. There is no feedback and the speed will vary if the golf cart is travelling up or down a hill. There are currently plans to implement a second speed control mechanism, which uses feedback from the wheel encoders in order to keep the golf cart at a steady speed.

Communication

The main MCU does not have enough power to processing the LIDAR and camera needed for autonomous mode. All this processing, as well as all other high level decision making is done inside a Linux box on the golf cart. This computer then communicates to the control board. There are currently two communication interfaces, a USART interface and a native USB-CDC interface.

USART Command Line

The USART communication interface runs over USB, but is converted to USART using a USB<->USART bridge on the Iris board. From the firmware's point of view, all communication over this interface is done through USART. The command line is fairly simple and is designed to be a human friendly communication interface.

All commands are started with a "[" character. The command name follows this. If arguments are required they are comma separated and written after the command name. Finally, a "]" ends a command name. At this point the firmware will process the typed command. For example, the command to set a debug LED is named "setLED" and takes 2 arguments. The first argument is the LED number and the second argument is the state, 1 for on and 0 for off. To set LED 6 on, the user would send the string "[setLED,6,1]". The command line also allows the use of backspace to deleted characters and colors the command to indicate if the command is valid.

Internally the commands are stored as an array of ConsoleCommand structures. The ConsoleCommand struct has three members, the command name stored as a string, the minimum number of arguments,

and a callback to call when the command has been entered. After the firmware has parsed the command name, it looks through this list, performing a string comparison on each entry. If the correct entry is not found an error is given. Otherwise the firmware checks the minimum number of arguments against the actual number of arguments provided. If the number of arguments provided meets or exceeds the minimum number of arguments, the provided callback is called, along with the arguments, processing the command.

USB Communication Interface

The native USB interface is designed to handle the bulk of the communication between the Linux computer and the control board. It is designed around command and response packets, where a command could be “Set FNR” or “Get Power levels”. A command packet may contain data if necessary. Once the control board receives a packet, it processes the packet and sends a response back to the Linux computer. The response may also contain data if necessary. The USB communication protocol was designed to have significantly less overhead than the USART command line, as well as have error checking through the use of a CRC8.

USB was chosen as the main communication protocol for a few reason. Firstly, it is easy to interact with from a PC. Using a USB-CDC interface, a PC can write to the port just like it was a serial port without any drivers required. Various libraries exist to make this easier, such as python’s pySerial, but a standard write syscall in C will also work. The main MCU also incorporates a USB Full Speed Physical Interface, which means that no external circuitry was required to get USB to work. Finally, ST provides a library to deal with the low level USB peripheral which eased development of a driver. The other choice for a communication protocol was Ethernet. Using Ethernet would allow debugging tools such as wireshark to be used to sniff communication. These types of tools exist for USB, but require external hardware and are expensive. Using Ethernet also required another chip to handle the physical protocol and would have provided another potential point of failure. For these reasons above, USB was chosen for the main communication interface. The protocol used is transport layer agnostic and could be used with both USB or Ethernet.

Future Improvements

As the firmware has been implemented, many new features and fixes to the current firmware have been suggested. As of now these fixes have not been implemented, but they are planned for the next couple of months.

It is yet to be seen how the switch to many service functions inside the main loop will affect performance. It has already drastically reduced the amount of debugging time needed to check for race conditions, but that comes at the cost of efficiency. In the coming months it will be necessary to profile the firmware in order to see if the service model has negative affect on the total efficiency of the firmware.

Often when the command line is being used the same command is issued over and over again in order to check the hardware. Currently there is no way to repeat the last command entered like one would

expect in a Linux command line. Added a history, even if small, would increase productivity when using the command line to debug.

Conclusion

With the current version of the firmware, all electronics necessary to drive and steer the golf cart are operational. This has allowed the team to perform milestone 1, where the golf cart is driven at a wall and stops to avoid hitting it. The detection of the wall is done using the LIDAR. As the golf cart approaches the wall, the Linux computer will send commands to lower the current speed. Once the golf cart is too close to the wall, the computer will switch the golf cart to reverse and back away from the wall at a fixed speed. This test control most of the golf carts electronics, with the exception of steering. A second test was performed where cone was placed in from of the golf cart. As the golf cart approached the cone it continuously checked to see if it could turn to avoid the cone. If it could, it would then turn to avoid the object. Otherwise it would stop. This tested all the electronics needed to drive the golf cart.

The next step is to create a test that integrates all the golf cart sensors, including the ultrasonic and hall effect sensors. This test is still in development and the team is excited to finally integrate all the golf carts sensors into a single test.

While there is still some work to be done on the firmware. I am satisfied with the current state of the firmware and believe that the remain team members will be able to finish up the last firmware tasks. I am also excited to see where the team is able to take the project in the future.

Appendix A – Control Board Schematics

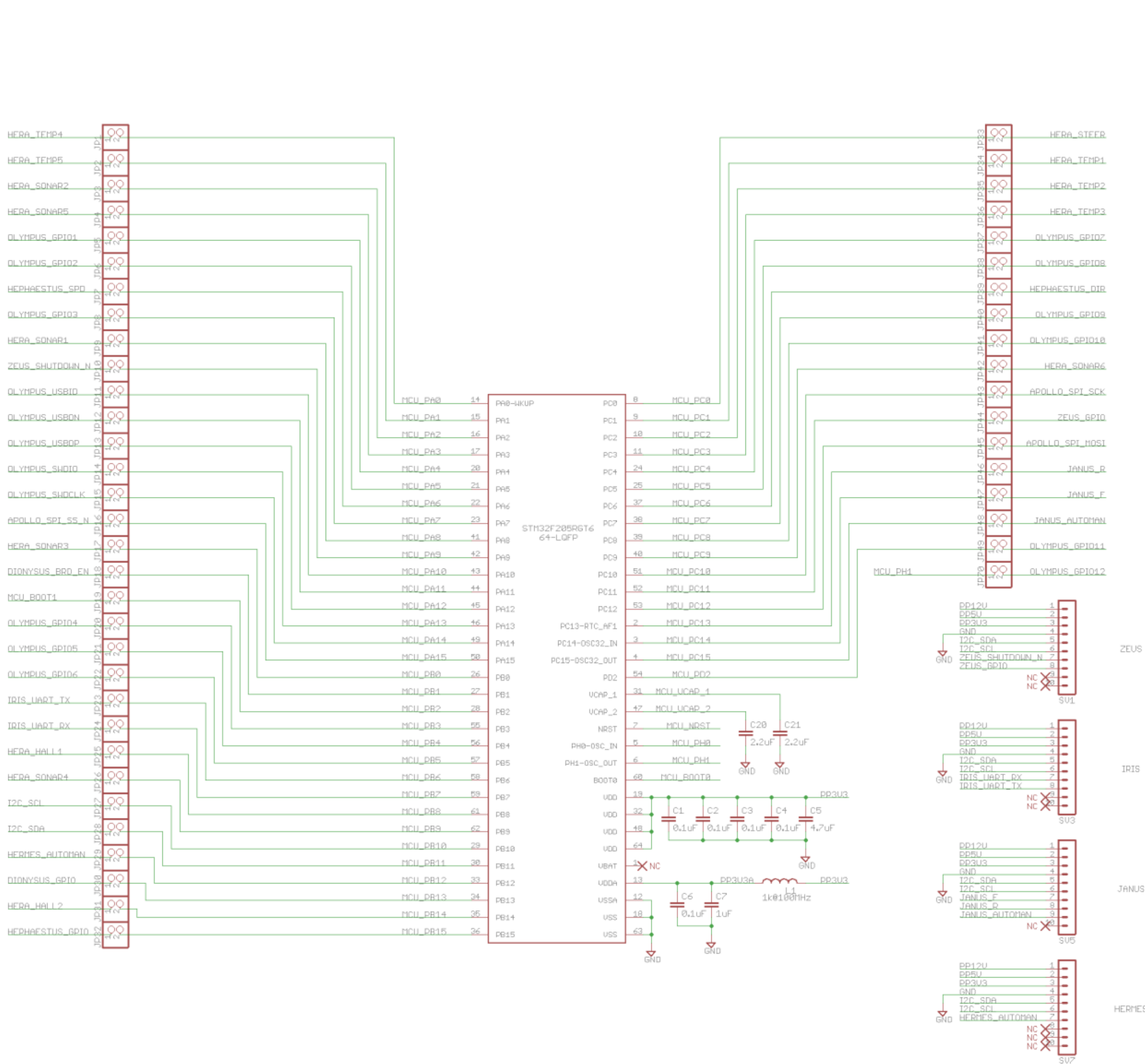

Figure A.1: Olympus Schematic Part 1

Figure A.2: Olympus Schematic Part 2

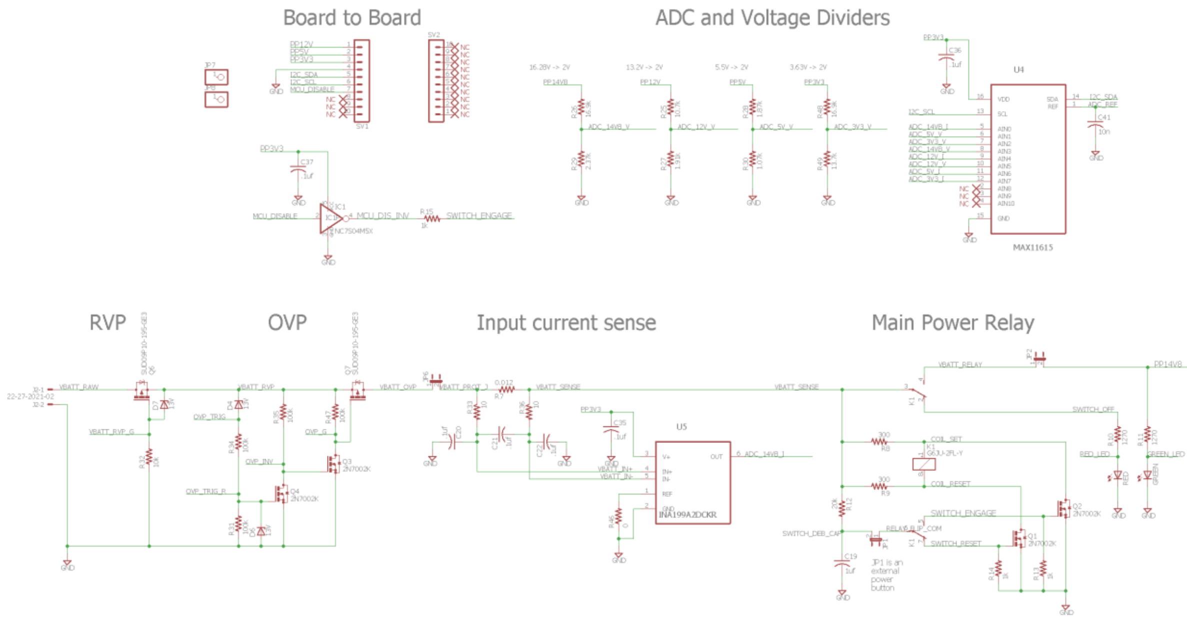

Figure A.3: Zeus Schematic Part 1

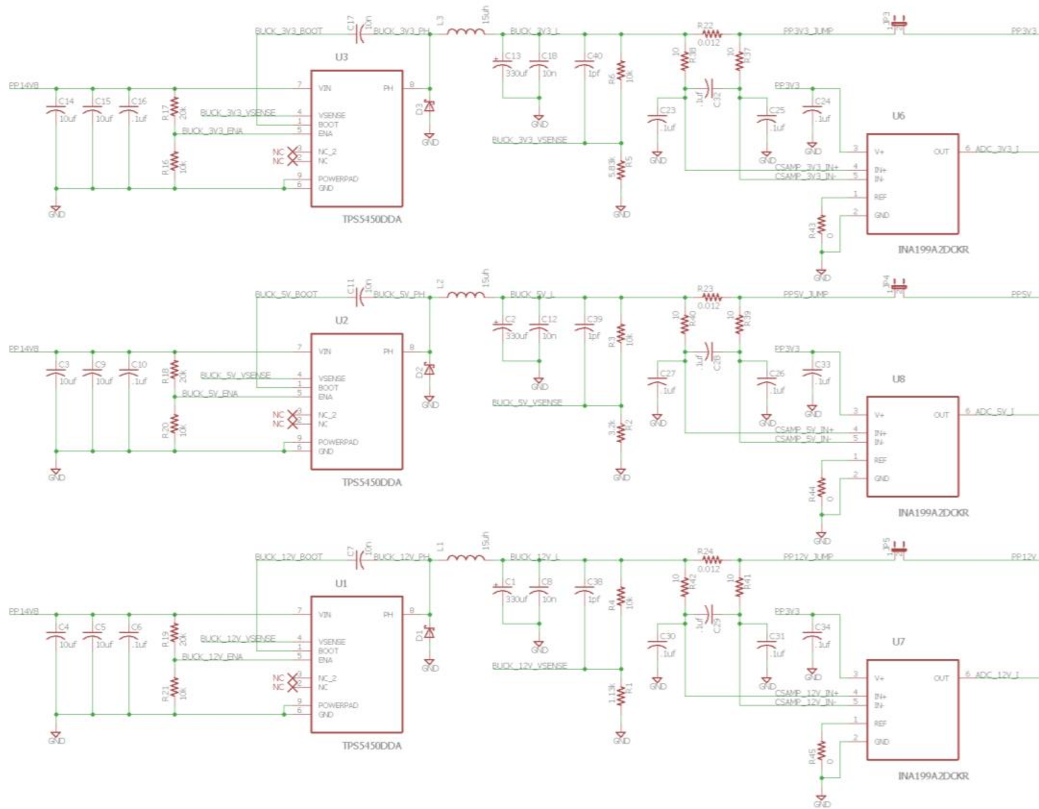

Figure A.4: Zeus Schematic Part 2

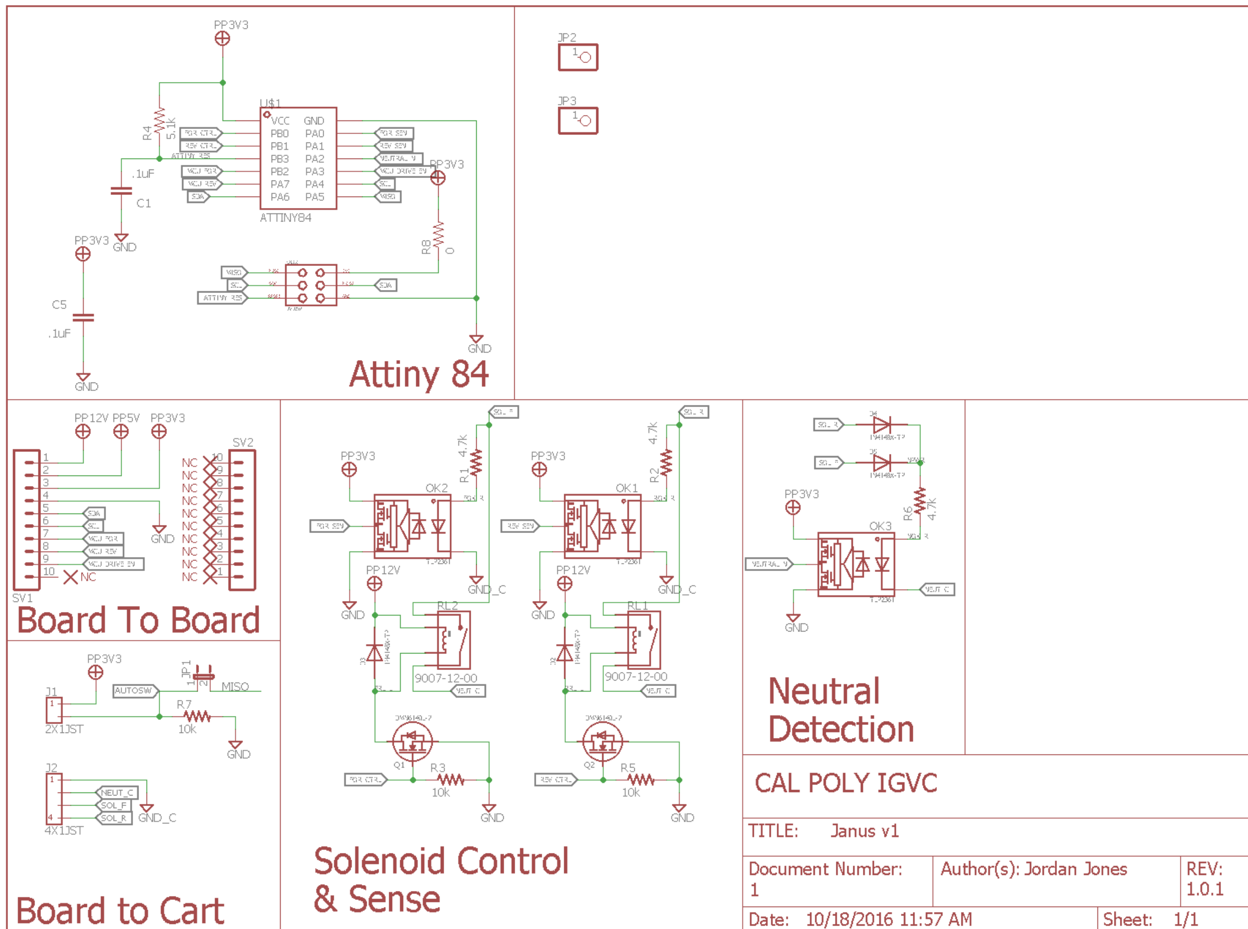

Figure A.5: Janus Schematic

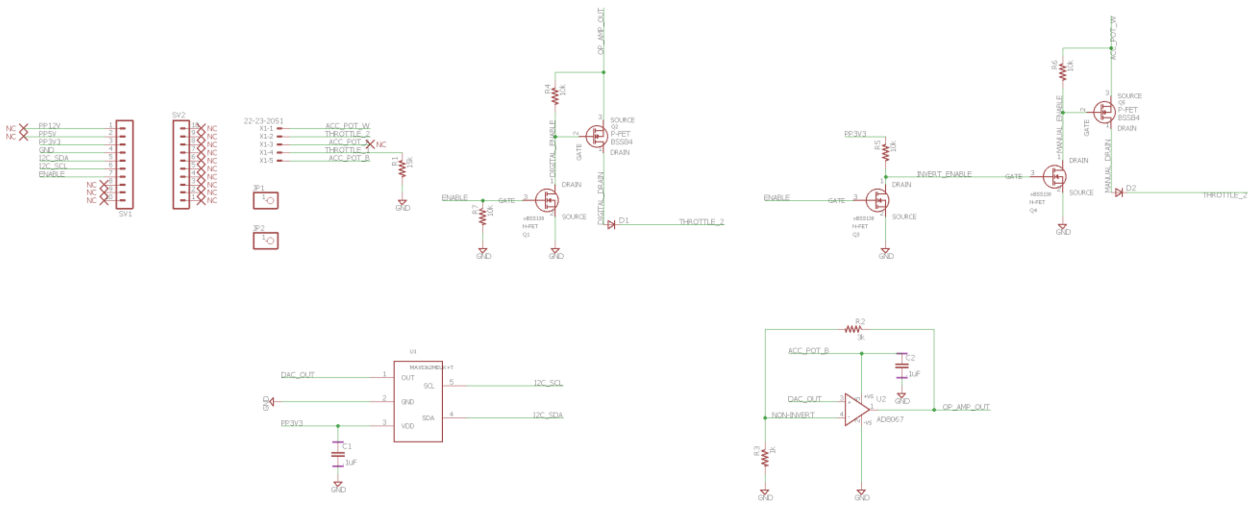

Figure A.6: Hermes Schematic

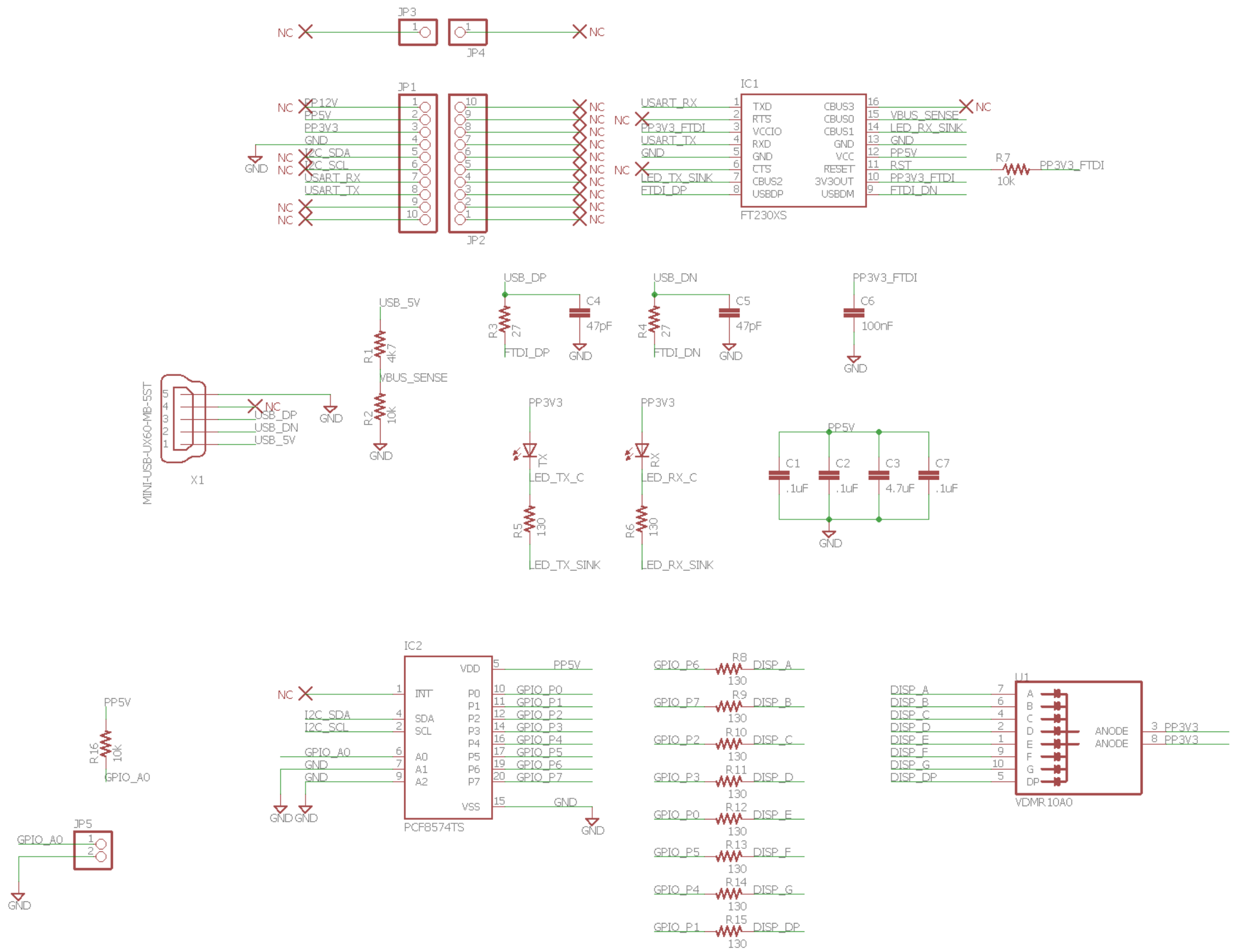

Figure A.7: Iris Schematic

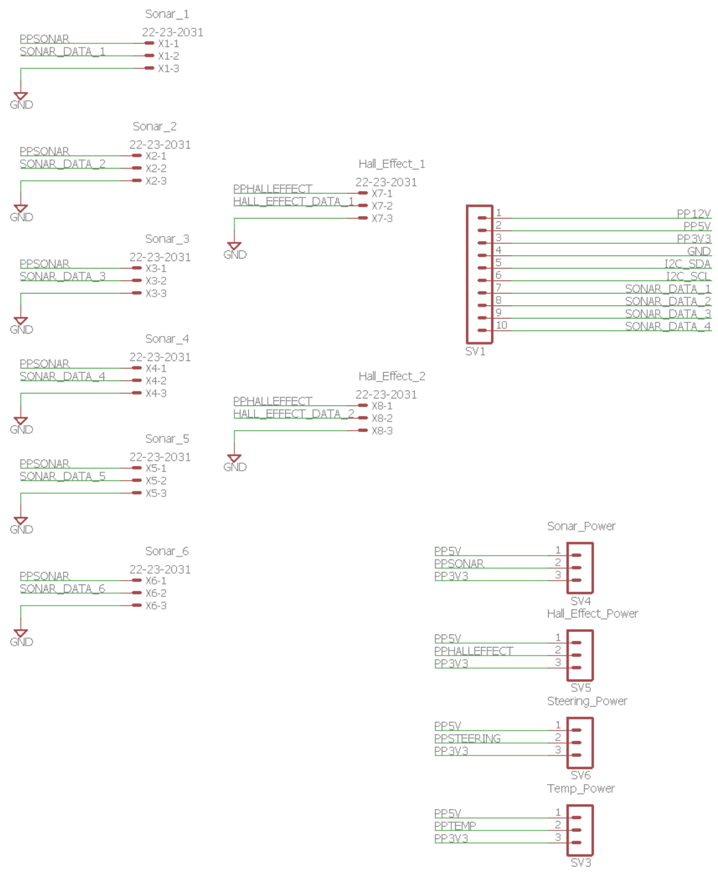

Figure A.8: Hera Schematic Part 1

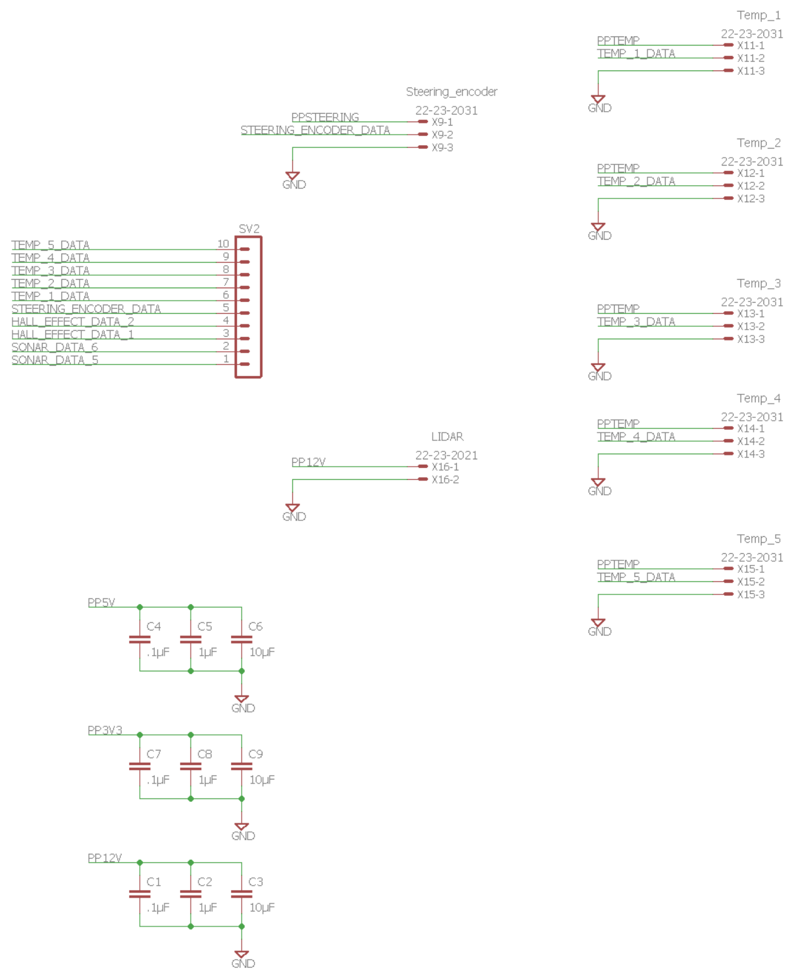

Figure A.9: Hera Schematic Part 2