Battery Pack Maintenance Manual

ECE/ME 492 | Spring 2020

Last Revision: 4/15/2020
Prepared by: Jon Abel, Clement Hathaway, Simone Khalifa,
Tim LaGreca, and Connor Winarczyk
Abstract

This maintenance manual aims to provide users detailed information to aid in debugging, fixing, understanding, and improving the accumulators. Included are schematics of electronics within the packs, links to design files, descriptions of firmware, procedures for working with the packs, and the principles of operation.
# Table of Contents

**Abstract**  
2

**High Level Pack Diagram**  
4

**Principles of Operation**  
5  
- High Current Path  
- AIRs  
- SMD  
- Battery Cells  
- Charging Path  
- CellMan  
  - Voltage Measurements  
  - Temperature Measurement  
  - Active Balancing  
- PacMan  
- PacMan PCB  
- Rear Panel  
- Side Control Panel  
- High Voltage/TS Active LED  
- Chassis Grounding

**PacMan Firmware**  
30  
- IDE & Setup  
- Programming  
- CANOpen  
- I2C  
- Object Dictionary  
- Display

**Maintenance Procedures**  
33  
- Test Points  
  - PacMan Test Points:  
- Fixing a Fuse  
- Recharging a Completely Dead Pack
High Level Pack Diagram

Figure 1 - High level diagram of two battery packs connected in series, including all internal wiring. (Note: This diagram is missing the 20A fuses between the charging power connector and PacMan. This is detailed in the Charging Path section below.)
Principles of Operation

High Current Path

The high current path within a pack runs from the TSV+ connector on the rear panel to the TSV- connector on the rear panel. Within the high current path are the 2 accumulator isolation relays (AIRs), the 300A Class T fuse, the segment maintenance disconnect (SMD), and the 16 LiFeMnPO4 cells, all wired in series. There are two methods of connecting these components: machined aluminum busbars and manufactured insulated flexible copper busbars. The aluminum busbars are used to bridge the terminals of adjacent cells and provide a mounting point for the flexible busbar. Segments of flexible busbar bridge the gaps between cell, SMD, fuse, and AIR terminals. This is all illustrated in Figure 2 below.

![Figure 2: High current path within a battery pack](image)

AIRs

Connected directly to each high voltage connector on the rear panel is an AIR, which is a beefy relay. These allow the always energized battery cells to be electrically isolated from the pack’s high voltage terminals. The packs use GIGAVAC GX14CAB normally-open relays. These relays use 24V coils which each draw 100mA in steady state to keep the contacts closed. Those coils are driven directly by the safety loop’s AIRS+ and AIRS- signals which are routed to each AIR through PacMan. When no voltage is present on AIRS+/-, the AIRs will remain open and the packs cannot provide high voltage.
SMD

Each pack has one SMD mounted on the rear panel. The purpose of an SMD is to provide a method for internally disconnecting high voltage for pack maintenance. The SMD in this design also is used to shut off power to PacMan when PacMan is operating off of the pack voltage. The SMD is a GIGAVAC HBD41 manual disconnect switch. When the switch is in the off position, the neighboring terminals of each cell stack are disconnected from each other, breaking the high voltage current path. Note that the SMD has the ability to be locked in the off position. This should be taken advantage of when the packs are left unattended, not in use, or open for maintenance.

Battery Cells

Each pack contains 16 LiFeMnPO4 prismatic cells connected in series. They are split into two segments of eight cells each. Each cell has a rated capacity of 60Ah, or 192Wh with a nominal voltage of 3.2V. The safely allowed voltage range for these cells is 2.5-3.8V, although it is expected that 3.6V will be the upper cutoff during charging.

Each cell has a positive and a negative terminal, with the positive terminal being surrounded with a red plastic ring. Each terminal has four M4 size screw holes used to mount the aluminum busbars. The aluminum busbar mounting holes are slotted and countersunk. The countersinks are to allow low clearance between each CellMan and the busbar for additional vertical room above the CellMen. The slots are to address manufacturing tolerances, since each cell is not exactly the same size and will have varying dimensions based on cell health and age.

As seen in Figure 1, the cells in each segment are arranged and connected in a “U” shape, with the segment positive and negative terminals on the same side of the stack. This allows all other pack components to be housed in one section on one side of the pack, and there are no wires being routed around or across cells. Each connection between cells is accomplished with an aluminum busbar. For the four cells that are at the ends of the segments, aluminum busbars with an M10 threaded hole are mounted to facilitate connection to the flexible copper busbars.

Charging Path

The charging path is displayed below in Figure 3. There are four 20A Littelfuse 0505020.MXP ceramic fuses in the charging path. Each is held in a Littelfuse 01500603Z heavy duty in-line fuse holder. The two fuses between the charging connector and PacMan are to protect PacMan in the event of short circuit on PacMan or before the next fuses. The two fuses between PacMan and the cells are to protect all components downstream of the cells while charging.
Each fuse holder comes with 19” of 14 AWG red wire lead. This lead is trimmed down, crimped and inserted into the proper connectors to form the charging cable assembly. This is displayed in Figure 4 below. The ring terminals are stacked and screwed on top of the flexible busbar pieces on the positive-most and negative-most cell terminals.
CellMan

Each pack contains 16 CellMen, which mount to the cells in a one to one configuration. The purpose of CellMan is twofold, it is responsible for making and reporting key measurements of the Cell it is mounted to, and for driving the segment’s active balance circuit. The most important measurements are of the cell’s voltage and temperature, which are used by PacMan to determine if a fault condition has occurred, and can also be used to calculate State of Charge.

Voltage Measurements

The Cell’s voltage is measured through a special kind of differential amplifier. Because all circuitry on the CellMan shares a ground with the lowest cell on the segment, it is possible for the Cell- and Cell+ terminals to be as much as 20 volts above the board’s power supply, preventing the use of most OpAmps. The AD8479 is a differential amplifier custom built for these kinds of operations, and uses a laser trimmed network of resistors to internally divide its inputs down to low voltages before amplifying them, giving it access to an extremely wide range of common mode voltages and an extremely high common mode rejection ratio.

Because the AD8479 has a fixed differential gain of 1, it is powered directly off of the 9V supply in order to maximize the range of voltages it can detect. In addition, it is given a 1V reference, allowing it to detect slightly negative voltages as well. The output of the AD8479 is then divided further by a 2:1 divider before being read by the ADC in the STM8.
In addition, CellMan is capable of measuring the absolute voltage of its Cell's negative terminal through an additional set of resistors. This information is used by PacMan to determine its position in the segment. Because there is much less need for precision with this measurement, discrete resistors can be used without issue.

To achieve compliance with the rules, the voltage measurement circuit is protected by two 2k resistors positioned near each cell terminal. A jumper is located between these resistors and the circuit to allow for the connection of a test circuit, which is used to test the board behavior against arbitrary cell voltages.

Temperature Measurement

Temperature measurement is done via a thermistor placed as close as possible to one of the cell’s terminals. The resistance value of the thermistor decreases at a predictable rate with a rise in temperature, and so the output of the above resistor network can be used to calculate cell temperature.

To maximize the thermal relationship between the thermistor and the cell, it is surrounded by a copper pour which is connected directly to the Cell+ terminal. Because this copper pour is unfused, additional resistors are added to the network in order to prevent catastrophic destruction of the board in the event that the thermistor and terminal are shorted together.

Resistance is calculated by the following equation:
\[ R_0 \cdot e^{(B_{25/85} \cdot (\frac{1}{T + 273.15} - \frac{1}{298}))} \]

Where \( R_0 \) is the resistance at 25 degrees celsius (2k), \( T \) is the temperature in degrees celsius, converted to kelvin by adding 273.15, and \( B_{25/85} \) is the thermistor’s B value which determines its sensitivity to temperature changes.

We can substitute this equation into the voltage divider equation with the values given to get a plot of output voltage vs temperature:

\begin{align*}
\text{Output Voltage vs Temperature} \\
\begin{array}{c|c|c|c|c|c|c|c}
\text{Temperature (°C)} & 0 & 25 & 50 & 75 & 100 & 125 \\
\hline
\text{Output Voltage (V)} & 0 & 1 & 2 & 3 & 4 \\
\end{array}
\end{align*}

While this plot is not linear, we can be considered approximately linear within a certain operating range in order to make calibration easier. We chose the range of 25C to 65C, which is the full temperature range the pack would be expected to operate in. The resulting linear approximation is:

\[ V_{out} = 0.0331216 \ \frac{V}{°C} \cdot T + 0.5542V \]

Which can be used to calculate temperature from voltage by the equation:

\[ T = 30.14 \ \frac{V}{°C} \cdot V_{out} - 16.64°C \]

Currently, this equation is hard coded into the CellMan firmware, but should ideally be stored in EEPROM to allow for easier modification of the temp sensing circuit.
A spreadsheet with the full analysis can be found here:
https://docs.google.com/spreadsheets/d/1xzh2DkGRcMngHCjEcCASSh_Z_XySEc_v28LqjC34Hjo/edit?usp=sharing
Active Balancing

CellMan is equipped with an experimental circuit for active cell balancing. Because each cell behaves slightly differently when charging and discharging, it is often the case that one or more cells will charge or discharge quicker than the rest, resulting in a significantly reduced overall capacity. An important feature of many Battery Management Systems, then, is to allow for the redistribution of energy between cells during operation. In this case, a cell can distribute its energy back into the segment it belongs to. A simplified diagram of this circuit can be seen below.
Distribution is done via an isolated flyback converter on each CellMan, which steps up the Cell’s voltage to that of the segment, generally from around 3.3V to 30V, although these numbers are expected to change during different kinds of operation. The stepped up voltage is then placed on the balance bus, a set of two wires which run between each CellMan on the segment. A set of jumpers on the highest and lowest CellMen in the stack connect the balance bus back to the segment’s positive and negative terminals. It is important that these jumpers be set correctly for the Active Balancing system to function properly, and it is possible that placing these jumpers in the wrong position can cause unexpected behavior or damage to the board.

In addition, it should be noted that, while JP1 can be ignored if the user does not intend to use the active balancing circuit, JP2 serves an additional purpose, and must be set even if no other part of the balancing circuit is installed. While the CellMen receive their power exclusively from a DC to DC converter on PacMan, their logical ground must be at the same potential as the segment’s minus terminal in order for voltage measurement to function properly. To achieve this, a series of net tie resistors (R11) connects the ground of each CellMan to the negative wire of the balance bus, which is connected to the negative terminal of the segment by JP2.

A more detailed schematic of the flyback converter can be seen below.
Firmware

CellMan uses an STM8 microcontroller to control its different subcircuits, perform basic calibration of data, and communicate with PacMan over I2C. CellMan is programmed using the SWIM protocol over the connector J1. A complete guide on programming a CellMan can be found in the Maintenance Procedures section.

Because a change made to the CellMan firmware needs to be propagated to up to 32 boards in the event of a full AMS system, it is worthwhile to keep the firmware as simple as possible. To keep with this goal, the CellMan source code was written with no external dependencies, and restricts its operation to a single update loop containing the following steps:

1. Read data from ADCs
2. Calibrate data to agreed upon units.
3. Update state of LED and Flyback Converter
4. Write Calibrated data to output registers.

All communication is handled by an Interrupt Service Routine, triggered whenever PacMan requests or sends data. A detailed description of the communication interface can be found in the section CellMan - PacMan Communication.
CellMan - PacMan Communication

Each CellMan is an I2C slave device, and can be controlled by a master, usually PacMan. All data reported to PacMan is sent over this interface, as well as commands that control the high level behavior of CellMan.

While CellMan on different segments are electrically isolated from each other, they occupy the same I2C bus, through 2 isolators on the PacMan board, and so their addresses cannot conflict.

Reading Data

When data is requested from CellMan, it will report it over the bus, one byte at a time, and always in the same order. An arbitrary number of bytes can be requested.

Most data is stored as 16 bit integers, and so is sent in 2 byte pairs, with the Least Significant Byte first, and the Most Significant Byte second.

Bytes are sent in the following order:

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Firmware Version</td>
<td>Cell Voltage LSB</td>
<td>Cell Voltage MSB</td>
<td>Temperature LSB</td>
<td>Temperature MSB</td>
<td>Position (^1) LSB</td>
<td>Position MSB</td>
</tr>
<tr>
<td>Unit</td>
<td>NA</td>
<td>mV</td>
<td>mV</td>
<td>1/10 °C</td>
<td>1/10 °C</td>
<td>Raw ADC Value</td>
<td>Raw ADC Value</td>
</tr>
</tbody>
</table>

Additional data can be added to the end of this as needed.

Sending Commands

PacMan is controlled by writing data to one or more 16 bit registers. Each write command should be 3 bytes long, a 1 byte address, followed by 2 bytes of data.

---

\(^1\) Position is a measurement of the Voltage of the Cell’s negative terminal with respect to the ground of its segment. Its value is currently only used to discover the Cell’s position in the Pack, and so no calibration is done.
Register addresses are organized as follows:

<table>
<thead>
<tr>
<th>Calibration (0x0X)</th>
<th>0x01: Temp Slope</th>
<th>0x02: Temp Offset</th>
<th>0x03: Balance Current Slope</th>
<th>0x04: Balance Voltage Slope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Balancing (0x1X)</td>
<td>0x11: Balance Enabled</td>
<td>0x12: Balance Duty</td>
<td>0x13: Balance Frequency</td>
<td>0x14: Target Voltage</td>
</tr>
<tr>
<td>Debug (0x2X)</td>
<td>0x21: Debug Mode</td>
<td>0x22: Unused</td>
<td>0x23: Debug LED</td>
<td></td>
</tr>
</tbody>
</table>

Addresses in yellow are reserved for planned features which are not implemented in software, addresses in red are deprecated features.
PacMan

PacMan is the top level accumulator management system (AMS) circuit board. It’s main functions are to track the safety status of the pack (cell voltages and temperatures), operate the AMS safety loop relay, facilitate pack charging, communicate with SCADA, and provide a user interface for configuring pack operation. This board was designed and created with KiCad version 5.1. The KiCad project and design files are located within the Github repository at https://github.com/Lafayette-FSAE/PacMan.

The PacMan board is split into two logical and physical sections. The side of the board that interfaces with low voltage systems like GLV power, the safety loop, and SCADA is the GLV side of the board, and the side of the board that interfaces with the cells and charging port is the TSV side of the board. Components on each side are electrically isolated from the other. Signals bridge that gap with isolators and DC-DC converters. In the schematic that separation is always illustrated.

PacMan can be powered from two sources - the high voltage input from the battery stack, or the low voltage input from the GLV battery. The high voltage power comes into PacMan in J1 and J2, which are Anderson PP30 30A power connectors. These connectors each carry a polarity of power to ensure good isolation distance and lower the probability of a short from the pack’s positive terminal to negative terminal. These power inputs are each fused at 500mA using Littelfuse 0453.500MR fuses in Littelfuse 0154.500DRL SMD fuseholders. PacMan should draw about 50-250mA from this input, depending on if the charging relays and CellMen are actively powered. The charging power input makes up the other halves of these connectors.
J3 and J4 are 6-pin IDC connectors for 28AWG ribbon cable. One 6-wire cable runs to each pack segment and connects to every CellMan in that segment. This cable carries 9V power, the I2C bus, and a segment-level reset signal. So, the CellMen are powered from PacMan, not each cell. J5 is a Molex Mini-Fit Jr. connector that mates with the charge detect cable coming from the charging port. Charge detection will be discussed below.

J6 is the RJ-11 connector that mates with the plug coming from the BBM-01 busbar current sensor. This cable carries +5V power and a differential analog signal representing the amount of current in the high voltage path.
Each connector listed above lives on the TSV side of the board since they directly interact with components that are in proximity to the battery cells and high voltage conductors. The connectors below live on the GLV side of the board. J10 connects PacMan to the safety loop input and output connectors on the rear panel. These connectors were chosen so that only one Molex connector has to be plugged into PacMan to connect both the input and output SLOOP ports. Likewise for J7, the GLV/CAN connector.

AIRS +/- are used on PacMan to enable the busbar current sensor, alert the ESP32 that the AIRS are closed, and close the AIRS. SLOOP1_IN is the safety loop signal coming into the pack and SLOOP1_OUT is the safety loop signal leaving the pack. These are on either side of the safety loop relay mentioned later on. SLOOP2 is the safety loop signal passing back through the pack from the last node in the chain.

GLV_24V and GLV_RTN are the 24V power rail from the GLV battery. CHS_GND is a connection to the frame and is tied to GLV_RTN at the GLV battery’s negative terminal connection point. CHS_GND is connected to the pack frame through ring terminals and wire from PacMan. CAN_L and CAN_H are the differential CANBus signals and CAN_SHIELD is the grounded metal shield located within the CAN cable.
J11 is a 20-pin IDC connector for 28 AWG ribbon cable that connects PacMan to the side interface panel. It carries button signals, display signals, and power and a signal path for the charging LED.

![Diagram of J11 connector](image)

J8 and J9 are 2 pin Molex connectors that route the AIRS +/- signal out to each AIR. The leads coming from each AIR should have 18-24 AWG Molex crimp terminals attached and then the mating housing for these connectors should be used.

![Diagram of J8 and J9 connectors](image)

As mentioned above, PacMan can be powered from both the battery stack and the GLV system’s 24V. To achieve that, the two rails are ORed together using low voltage drop Schottky diodes D5 and D6. This power rail labeled LV_24V (it is really ~23.5V due to the diode drop) is considered the “master” power bus for the board. DC-DC converters use that 24V to generate each 9V rail for each segment’s CellMan bus, 5V for the high voltage side of the board, and 5V for the low voltage side of the board.

Note that the three high voltage power rails are all isolated from each other. Each segment rail contains a jumper which ties the respective RTN_SEG node to a “segment” ground within the stack, for the benefit of CellMan’s circuitry. If these were shared power rails on PacMan, then there would be a shorted segment (bad). The high voltage side of the board has its own ground as well, so that there is no requirement for the lowest segment (which shares a ground with this high voltage section) to be plugged into a specific connector on PacMan. In other words, either segment can be plugged into either 6-pin segment connector.

Decoupling caps are included for every converter for stability per datasheet recommendations. R34 provides a minimum load of 10mA for U12 in order to maintain voltage regulation. An additional diagram below helps illustrate PacMan’s power scheme.
The pack’s battery stack voltage, which ranges from about 40-60V, is converted to 24V using an isolated DC-DC converter with an allowed input range of 18-75V and maximum current output of 416mA. To improve the accuracy of the pack’s state of charge (SOC), a shunt resistor is used to measure the current being drawn by PacMan from the cells. The voltage generated
across the shunt is measured by an LTC4151 power monitor chip, which also measures the full battery stack voltage and communicates both values back to the ESP32 via the I2C bus. Decoupling caps C14 and C15 are included per the datasheet for stability.

This LTC4151 chip also has an auxiliary ADC input. An INA592 differential amplifier takes the differential output signal from the BBM-01 busbar current sensor (roughly 0.5-4.5V centered about 2.5V), generates a single ended signal reference to ground (note the output gain of 0.5) and feeds the voltage into the ADC, where it is fed back to the ESP32 via the I2C bus. This allows the ESP32 to calculate the high voltage current being drawn from the pack, in the neighborhood of 50-300A. Decoupling cap C16 is included.
PacMan has two normally open, 30A TE Connectivity T9AS1D12-24 relays on it in order to isolate the battery terminals from the interface panel and charging connector when the pack is not actively being charged. The coils operate at 24V and together draw about 85mA in steady state to keep the contacts closed. An active-high N-channel MOSFET is used to control the closing of the relays. This logic-level FET is driven directly from the ESP32. It also controls an LED charging indicator on PacMan, and an LED indicator on the side panel. When Q2 conducts, the voltage drop across R49 allows the P-channel MOSFET Q3 to conduct, sending 24V to the LED panel mount indicator on the interface.

To enable tracking pack SOC, a current sensor is included on PacMan in the path of charging current. The ACHS-7122 is a +/- 20A rated Hall Effect current sensor that outputs an analog voltage between 0.5V and 4.5V corresponding to -20A and 20A. This output is divided down by resistors R24 and R25 to levels suitable for the ESP32. Capacitor C40 is included to add filtering to the ESP32’s ADC input to help reduce noise. Continuously sampling the current during charging enables the use of SOC algorithms as simple as Coulomb counting. Decoupling/filtering caps C12 and C13 are included. Flyback diode D4 is included to protect the relays and transistor from inductive voltage spikes.
Every CellMan and most devices on PacMan use I2C to communicate with each other, so there is a single I2C bus that runs from PacMan to each CellMan chain. In order to maintain galvanic isolation between PacMan board sections, Si8600AC I2C isolators are used for each section that requires access to the I2C bus (all of them). Each separate section of the bus needs pull-up resistors on each I2C line. Since the ESP32 needs 3.3V GPIO input, the GLV side of the bus is pulled up to 3.3V. Each high voltage side of the I2C bus is pulled up to 5V, as every device on that side runs off of 5V. Decoupling caps are included for each chip.

It is very helpful for the ESP32 to know when the charging connector is plugged in, as it allows the ESP32 to completely control charging without any required input from the user. In order to accomplish this, there is a jumper in the Anderson Pak Plug Shell that is internally shorted. When the connector is plugged in, the pins CHRG_DETECT+ and CHRG_DETECT- are shorted together. R29 and R30 act as isolation resistors of sorts, since this whole assembly of components resides on the high voltage side of PacMan. C23 helps filter out bouncing from plugging/unplugging the connector. When CHRG_DETECT+/- are shorted, the gate of P-channel MOSFET Q5 is pulled close to ground, allowing current to flow through the LED of optoisolator ISO2 and current limiting resistor R31. As current flows through the LED, pin 4 of ISO2 is pulled to ground from 3.3V, and the ESP32 is now aware that the charge connector is
 plugged in. R48 and C42 are included in the event that more analog filtering is needed to prevent erroneous interrupts from triggering.

Since the BBM-01 busbar current sensor draws an insignificant amount of current, it is nice to only provide power to it when it would need to be sensing current, i.e., when the AIRs are closed. When 24V is across AIRS+/-, pin 4 of ISO4 is pulled low and current can flow from the 5V TSV side supply through Q4 to the current sensor.

PacMan has the ability to reset each segment individually (resetting a segment means resetting each CellMan on that segment). Since the segments are isolated, optoisolators ISO3 and ISO7 enable resetting. The reset pin on CellMan is an active low, so a pull-up resistor for the reset line is included on PacMan.

The ESP32 DevKitC is a microcontroller development kit that is easy to plug in and swap out from PacMan, thanks to its pin headers. It connects to a computer via a Micro-USB cable. Of its 38 total pins, 32 are usable to us (to avoid conflicting with the flash memory SPI bus) and of those 32, we found the maximum usable number of pins to be 27. SENSOR_VP and SENSOR_VN are connected to the internal ADC’s power regulator, leading to conflicts. TXD0 and RXD0 and used for serial output and are routed through the USB port on the dev kit. IO0 is a bootstrap pin that can cause issues when used improperly. So, about as many pins on the ESP32 are being used as possible.
A watchdog is included on PacMan so that in the event of a microcontroller crash, the watchdog can automatically reset the ESP32. Decoupling caps C2, C3 and C41 are included.

A major function of the AMS is to open the AMS safety loop in the event of cell damage, cells exceeding set performance thresholds, or other catastrophic failure. The ESP32 controls the normally open 24V coil G5Q-1A4-DC24 safety loop relays through the NMOS transistor Q1. Flyback diode D1 is included to protect the relay and transistor from inductive voltage spikes.

An MCP2551 CANBus transceiver chip takes CAN formatted serial data from the ESP32 and drives the output CANBus appropriately, and vice versa for receiving data. C5 is a decoupling cap. **R6 should normally be left unpopulated. It acts as a terminating resistor for debug purposes only.** R1 and C4 act as a filter for the CAN shield connection, and R4 sets the slew rate of the MCP2551. **Note:** R1, C4 and R4 were all left unpopulated this year. R1 and C4 were not confidently sized, and R4 is unnecessary unless you desire to slow the slew rate of the transceiver to decrease high frequency content on the CANBus.
R9, R11, R13, R15 and R17 act as pull-up resistors for the five buttons on the interface panel. The remaining components act as RC low-pass filters to remove the effect of bouncing on the buttons. Note that the resistors are sized to still pull the buttons close enough to ground to register as a logic low on the ESP32’s pins.

Since all available pins on the ESP32 were already used, an I/O expander was included for additional GPIO capabilities. The MCP23008 is an I2C device with 8 input/output channels. Included on it are pins to drive the two misc. LEDs on PacMan, each segment reset signal, and 4 inputs which monitor the status of the safety loop and GLV power. C6 is a decoupling cap. The 4 inputs can be configured to generate interrupts on change, so that IO_INT will alert the ESP32 when an interrupt has occurred. This allows the ESP32 to be aware of any safety loop or GLV power changes without explicitly polling the MCP23008 continuously over the I2C bus.
The **TLP293-4** is a 4 unit optoisolator IC. The 4 isolators are used to measure AIRS+/-, SLOOP1_IN, SLOOP2, and GLV_24V. Isolators were used to avoid any sort of safety loop interference or shorts from affecting the ESP32. The outputs of each isolator are all pulled up to the 3.3V rail, making all of them active-low. While measurements of AIRS+/-, SLOOP1_IN and SLOOP2 are purely informational to send to SCADA to assist in debugging the safety loop, the measurement of GLV_24V allows the pack to determine when it is connected to the rest of the car, which influences its behavior when charging (see the competition rules for charging).
There are two ambient temperature sensors on PacMan, one for each side. This decision was made because when installed, there will be a Garolite insulating panel mounted above PacMan to isolate the TSV side from the GLV side. This could lead to a situation where the TSV side is overheating due to charging or cell/conductor overtemperature and the GLV temperature sensor does not detect that temperature change due to the barrier. So, each side has an MCP9804 I2C temperature sensor.

The last component on PacMan is an I2C real time clock (RTC). The BQ32002 is powered both by the 3.3V rail and a coin cell battery so that it may remain operating when the system is turned off. A 32K crystal was sized for the IC and load capacitances were added to help stabilize the crystal (the designer is not exceedingly confident in these values, so do with them as you will).

PacMan PCB

There are a few things to mention about PacMan’s PCB. Notice that the bottom layer of solder mask over the copper pours in the charging path was removed. This is to allow adding solder and copper braid to those traces. If a length of copper braid/solder wick (there should be a
newly purchased spool of solder wick with the PacMan parts) is soldered to the traces, that should increase the amount of current those traces are able to handle. This is important, as the packs can be charged at up to 20A, and the current traces likely could not handle more than 10A or so before heating up and affecting nearby components.

It should be noted that care was taken to adequately space out components on the TSV side of the board. Important spacings are indicated by dimensions annotated on the Dwg.User layer. The competition rules set limits on how close certain conductors are allowed to be. It is always better to exceed those spacing requirements as much as possible and in as many places as possible, so that there is not a surprise issue with TS spacing requirements at the competition.

Rear Panel

The rear panel connects a pack to the rest of the system. There are 6 connectors on the back: TSV+, TSV-, SLOOP IN, SLOOP OUT, GLV/CAN IN, GLV/CAN OUT. TSV+/- carry the high voltage. These connectors are Amphenol 1POS PowerLok receptacles. The SLOOP IN and SLOOP OUT connectors are 4-pin Deutsch DT flanged panel mount receptacles with part number something like DT04-4P-CL06. The GLV/CAN IN and OUT are 6-pin Deutsch DT
sealed panel mount receptacles with part number something like DT04-6P-CL09. At the time of writing, the pinouts were never confirmed, however pinouts exist on the system-level wiring diagram from Interconnect.

Side Control Panel

Each pack has a control panel that acts as an interface for the user on its side. Contained in this panel is a high voltage LED, a charging LED, a charging port, a USB port, a display, and five buttons to control the display. The purpose of the high voltage LED is to indicate to anyone observing the car when there is TSV running through the packs. The charging port is used for charging the cells in the pack, and when it is plugged into, the charging LED will light to indicate that the packs are being charged. The USB port may be used to reprogram the microcontroller (ESP32) on PacMan without taking apart the packs. The display is a user interface that indicates the state of the packs, possible faults, and allows the user to program configuration parameters. In order to do this, they may use the button located on the panel next to the display.

High Voltage/TS Active LED

On the interface panel is a red LED indicator that illuminates whenever the DC voltage across the vehicle side of the AIRs output is greater than 19V. This indicator is powered directly by a chassis mount PQDE6W-Q48-S24-T DC-DC converter. It has an input voltage range of 19-75V, so whenever the input voltage is at least 19V it will output 24V directly to the LED indicator. As the competition requires voltages of at least 30V, this exceeds that requirement. Protecting the input of the DC-DC converter are 500mA Littelfuse 0229.500MXP fuses housed within a Littelfuse 02540001Z chassis mount cartridge fuse holder. The fuse holders are connected on one to the vehicle side of the AIRs with ring terminals and 16-20AWG wire. On the other side, 16-20AWG wire runs to the input of the DC-DC converter, which is mounted above PacMan. The fuse holders are mounted directly beside each AIR.

Chassis Grounding

The rules require that all metal enclosures mounted on the frame have a resistance less than 0.3ohms to ground/GLV_RTN. To help lower that resistance, the battery pack enclosure is grounded to CHS_GND through PacMan, in addition to the frame. To accomplish this, the mounting holes on PacMan’s GLV side are connected to the CHS_GND signal coming through the CAN/GLV line. The pack enclosure should be electrically connected to those mounting holes using ring terminals and green wire.
**PacMan Firmware**

PacMan firmware uses three types of communication to send and receive data. The PacMan is connected to a CAN Open Node Bus where it may share information with the rest of the car. It uses I2C to read data from and interact with the CellMen. All of this data is written to the Object Dictionary which also houses configuration variables. The display is operated through SPI communication and draws values from the Object Dictionary.

**IDE & Setup**

1. Install the [Arduino IDE](https://www.arduino.cc). Tested on version 1.8.10
2. Add ESP32 board to the Arduino IDE by adding this Additional Board Manager URL: https://dl.espressif.com/dl/package_esp32_index.json. A tutorial is located [here](#).
3. Select the ESP32 Dev Module in the boards list - Though most of the common boards here listed here will be compatible
4. Clone this Github repository. A tutorial for git can be found [here](#). We recommend also using Github Desktop or IDE with Github support like Atom or Visual Studio Code; the latter of which can also be used to easily programme and upload to the board if you are not a fan of the Arduino IDE.
5. Open up the file: `PacMan_CANOpen.ino` located in the `PacMan_CANOpen` directory in the repository. This should open in the Arduino IDE if installed correctly

**Programming**

1. Connect the ESP32 Dev Board to the computer via the microUSB port
2. Select the proper serial port from the "Tools" drop down and set the baud rate to 115200 (isn't it funny how all the baud rates are multiples of 60 ;))
3. Click on the compile and upload button (The right-facing arrow on the upper-left of the IDE)
4. Open the Serial Monitor by clicking on the Magnifying glass in the top right corner or going to "Tools" -> "Serial Monitor" after the computer has compiled and uploaded the code

You can also programme the ESP32 board with a precompiled image (binary file) as well:

1. Install ESPtool (esptool.py) using pip install esptool
   1. This requires Python which is installed by default on most Linux Distros (including the Raspberry Pi), but otherwise can be downloaded and found [here](#)
2. Plug in the ESP32 on the PacMan board via its USB cable
3. Locate it's serial port:
   1. [Debian (Ubuntu/Raspbian) Tutorial](#)
2. Mac OS X Tutorial
3. Windows Tutorial (First answer)

4. In Terminal / Command Prompt type (excluding bracketed information):
   1. esptool [or esptool.py] --chip esp32 erase_flash
   2. esptool [or esptool.py] --chip esp32 --port [port you found in part 3] write_flash 0x1000 [Latest binary file]

5. Wait for flashing to complete and you should be good to go!

CANOpen

1. Make sure the CAN bus wires are properly connected
2. Make sure SCADA is receiving data from the PacMan
3. Set DEBUG flag in the code to True
   a. Follow Serial output and make sure
4. Make sure the message queue is being emptied (look in the serial terminal to see if we stop queueing messages - AKA the queue is now full)
   a. This means there is a communication problem, usually physically
5. Make sure the CAN driver was installed properly (serial output)
6. Try sending CANopen commands to the PacMan using the RaspberryPi w/ a CAN Hat 2 installed and cansend and candump from can-utils
7. Make sure the display is showing appropriate data since CANopen and the display core share the same data structure
8. Using a RaspberryPi and cansend & candump, send the command:
   60A#5102300900000000 or 60A#5102300100000000 to access the voltages of the first cells. You Should expect a response like: 58A [8] 4F 02 30 00 10 00 00 00
9.

a. The 60A is the address with A being the number you preprogrammed the address to be.

b. The 51 tells the system you are asking for an upload ensuring it will send data and not store it.

c. 0230 is the address that you used with 3002 being the actual address. This specific address was for the cell voltage.

d. 01 is the sub address for the accessed data pointing to the exact cell.

e. 58A is the system's reply it is followed by the exact same 4 byte inputs that were inputted then it follows with 2 bytes that contain the system's reply. 0010 would be the reply in the example above.

f. 70A is the wake up call for the system. The system will output this on wake up so this can be ignored though it should usually only send one message.

g. 08A is a routine message to the system. The purpose is not known at the moment. So you can figure that out yourself.

h. Remote access is a message you receive when the data you are trying to pull is not able to be accessed. This is usually because another system is using it at that moment or because it is not data you can access.

I2C

1. Make sure that both segments are connected to the individual Segment connectors on the PacMan board.
2. Make sure the CellMen are getting power delivered by this connection
3. Make sure the PacMan can see the CellMen devices by resetting the ESP32 and viewing the terminal output (The LEDs on the CellMen boards should flash too)
4. Set DEBUG flag in code to true
   a. Observe related outputs such as collecting data from cellmen and making sure things do not hang on this stage
   b. Also observe that CellMen devices have been detected in setup
5. Connect the digital input on the oscilloscopes in the lab and put the triggering on bus and select I2C and obverse that there is a response from the CellMen and a Packet being sent by CellMen in accordance to the I2C Protocol

Object Dictionary
The Object Dictionary has been adopted from a GitHub Repository by robincornelius. If looking to add a variable to the Object Dictionary: Download the EDS Editor linked above. Open the .XDD file linked under “Object Dictionary” here. Navigate to the Object Dictionary Tab. Right click anywhere in the box called Manufacturer Specific Objects. Click Add New Object. Add your new variable. If looking to edit a variable that already exists, follow these same steps but rather than right clicking, choose the variable you are editing and make changes accordingly. To export your .c and .h files, click File and choose Can Open Node c/h. Once these are exported, navigate to the bottom of CO-OD.c and add corresponding strings to your new variables.

Display
The main screen for the display uses a background image that may be found here. This image was created using Microsoft Paint and then saved as a .bmp file. In order to configure an image to be used on the display: Download IMG2LCD. Open your .bmp image. Set the following parameters: Output File Type- C array, Scan Mode- Vertical Scan, BitsPixel- monochrome, Max Width- 296, Max Height- 128. Check Reverse Color. Once this is done, click Save. Open your new.c file, copy all of the contents, and paste them into your BitmapGraphics.h file.

Maintenance Procedures

Test Points
Test points provide a quick and easy way to verify system voltages. They are very helpful in determining if your power rails are operating correctly.
PacMan Test Points:

<table>
<thead>
<tr>
<th>Test Point Name</th>
<th>Schematic Name</th>
<th>What It Is</th>
<th>What It Should Be (Probably redundant)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP1</td>
<td>5V_SEG-</td>
<td>5V rail for Segment- bus</td>
<td>5V wrt RTN_SEG-</td>
</tr>
<tr>
<td>TP2</td>
<td>RTN_SEG-</td>
<td>Segment- bus ground point</td>
<td>0V wrt RTN_SEG-</td>
</tr>
<tr>
<td>TP3</td>
<td>9V_SEG-</td>
<td>9V rail for Segment- bus</td>
<td>9V wrt RTN_SEG-</td>
</tr>
<tr>
<td>TP4</td>
<td>LV_RTN</td>
<td>Low voltage ground point</td>
<td>0V wrt LV_RTN</td>
</tr>
<tr>
<td>TP5</td>
<td>LV_5V</td>
<td>5V rail for low voltage side</td>
<td>5V wrt LV_RTN</td>
</tr>
<tr>
<td>TP6</td>
<td>LV_24V</td>
<td>24V rail for low voltage side</td>
<td>~23.5V wrt LV_RTN</td>
</tr>
<tr>
<td>TP7</td>
<td>GLV_24V</td>
<td>24V rail from GLV battery</td>
<td>24V wrt LV_RTN</td>
</tr>
<tr>
<td>TP8</td>
<td>SDA</td>
<td>Data line for I2C bus</td>
<td>A square wave transmitting the specified I2C bytes^2</td>
</tr>
<tr>
<td>TP9</td>
<td>SCL</td>
<td>Clock line for I2C bus</td>
<td>Should be a consistent square wave when the I2C bus is active^1</td>
</tr>
<tr>
<td>TP10</td>
<td>5V_TSV</td>
<td>5V rail for high voltage side</td>
<td>5V wrt RTN_TSV</td>
</tr>
<tr>
<td>TP11</td>
<td>RTN_TSV</td>
<td>High voltage side ground point</td>
<td>0V wrt RTN_TSV</td>
</tr>
<tr>
<td>TP12</td>
<td>9V_SEG+</td>
<td>9V rail for Segment+ bus</td>
<td>9V wrt RTN_SEG+</td>
</tr>
<tr>
<td>TP13</td>
<td>5V_SEG+</td>
<td>5V rail for Segment+ bus</td>
<td>5V wrt RTN_SEG+</td>
</tr>
<tr>
<td>TP14</td>
<td>RTN_SEG+</td>
<td>Segment+ bus ground point</td>
<td>0V wrt RTN_SEG+</td>
</tr>
</tbody>
</table>

Fixing a Fuse

Before changing a fuse, confirm that a fuse is blown. Measure the resistance of the fuse with an ohmmeter. Depending on the fuse, if the resistance is anything higher than a couple ohms, it’s likely blown or damaged. For the Littelfuse Nano fuses which are mounted in the Littelfuse surface mount fuse holders, use a pair of pliers to grip the sides of the fuse and pull directly out. Replace with a fuse with equal or comparable current rating and DC voltage rating.

^1 Talk to an advisor about probing the I2C bus with a communications module for an oscilloscope

^2 Talk to an advisor about probing the I2C bus with a communications module for an oscilloscope
For the ceramic fuses housed within the in-line fuse holders, twist the cap of the fuse holder while pushing in to remove the cap. Remove the fuse, replace it, and push the cap back on and twist to lock it in place. For the cartridge fuses located next to the AIRs, just pop the blown fuse out and insert a new fuse.

**Recharging a Completely Dead Pack**

One of the motivations behind being able to power PacMan from two sources at the same time was enabling dead pack charging/operation. If the pack is completely dead, then there will not be sufficient voltage at the power input to PacMan for the ESP32 to boot, initiate and control plug and forget dead pack charging, or for the DC-DC converters to power the CellMen. However, since GLV_24V can also be used to power PacMan (and all of the CellMen), the AMS can be turned on with a bench power supply feeding 24V into the GLV/CAN connector’s GLV_24V pin and the ESP32 can start a controlled recharging of the cells, all without having to open up the packs for true maintenance.

**Programming CellMen**

Materials:
- STLinkV2 (Dedicated STM programmer)
- A way to connect the CellMan board to the Programmer
- A computer with USB (Linux is easier, but a windows machine with cygwin is possible)
- The following software:
  - Stm8flash -- [https://github.com/veludouyt/stm8flash](https://github.com/veludouyt/stm8flash)
  - (In my case, it was necessary to compile these from source)

1. Connect the Power, GND, Reset, and SWIM pins on the J1 connector of CellMan to their corresponding pins on the Programmer, and the Programmer to the USB port of your computer.
   a. As of right now, no dedicated cable exists for this purpose. It was my intention to make one, but I find myself unable to at the time of writing this. A set of female to female jumper wires can be used instead.
   b. On later iterations of CellMan, the pins on the J1 connector are not populated. A good connection can be made temporarily by pressing a set of male pins against the holes and plugging the jumper wires or cable into these pins
   c. If CellMan is receiving external power, the Power pin can and should be omitted.
2. Clone the CellMan firmware source code from Git and make any desired changes
   a. [https://github.com/Lafayette-FSAE/CellManFirmware](https://github.com/Lafayette-FSAE/CellManFirmware)

3. Once you are confident that SDCC and STM8flash have been installed correctly and are
   in your path, navigate to the CellManfirmware folder in your terminal and run the
   following commands.

   # make sure both tools are installed correctly
   $ stm8flash
   $ sdcc

   # compile and flash the code
   $ make
   $ make flash