Power System

Reference

V4.0

R06

Revision history

Author	Version	Revision	Date	Changes summary
Arion Zimmermann	PS V1.0	R01	12/2020	Initial documentation uploaded
Arion Zimmermann	PS V2.0 (Pollux I)	R02	06/2021	Rewritten documentation
Arion Zimmermann	PS V2.1	R03	08/2021	Updated documentation
Arion Zimmermann	PS V3.0 (Pollux II)	R04	08/2022	Rewritten documentation
Arion Zimmermann	PS V4.0 (Pollux III)	R05	08/2023	Rewritten documentation
Arion Zimmermann	PS V4.0	R06	10/2023	Updated documentation

Table of contents

1.System requirements

The power system is a standalone subsystem in EPFL Xplore's rovers. It is the subsystem which possess by far the least number of interfaces with the rest of the rover. Nevertheless, these few interfaces are critical to the proper functioning of the rover and a single failure of these interfaces can lead to a mission failure.

When defining the power system, the System's Engineering workflow could be decomposed in the following manner:

1) Define the Concept of Operations (ConOps).
2) For each subsystem, define modes of activity and assign them to the ConOps phases.
3) Assess the power needs for each subsystem for each mode of activity. Make sure to minimize the number of different voltages when assessing the subsystem's needs (trade-offs must be made!).
4) Compute the power and energy budget.
5) Write **testable** requirements accordingly.
6) Deliver the requirements to the power system designers.
7) Write test procedures to validate the requirements.
8) Deliver the test procedures to the power system testers and ensure that all requirements are fulfilled. Make trade-offs if needed.
9) Integrate electronically the power system as soon as possible with the other subsystems. This makes sure that the system requirements represent well the power/stability needs of the subsystems.
10) Benchmark the system by measuring its efficiency and temperature increase across many input/output voltages and load conditions.
11) Integrate structurally the power system.

1.1. Nominal requirements

The design of the power system is derived from a set of requirements defined by the systems engineering authority. These are system constraints necessary for the proper functioning of the Kerby 2023 rover. Table @ lists the requirements proposed by the systems engineering team.

ID	Title	Description
XP_EL_001	5V rail	The 5V channel shall be able to provide up to 3.5 [A].
XP_EL_002	12V rail	The 12V channel shall be able to provide up to 3.5 [A].
XP_EL_003	15V rail	The 15V channel shall be able to provide up to 3.5 [A].
XP_EL_004	24V rail	The 24V channel shall be able to provide up to 3.5 [A].

These high-level requirements set a lower-bound performance on the power system and define the smallest set of necessary functionalities. Nevertheless, in the prospect of reusing the same power supply design for future projects in robotics, an extended set of requirements was defined by the power system design authority. The extended

requirements, listed in the following tables, enforce the design of a modular, scalable, and redundant power system.

1.2. Extended requirements

1.2.1. System requirements

ID	Title	Description
XP_PWR_GEN_001	Power system definition	The power system consists of identical power supplies.
XP_PWR_GEN_002	Power system configuration definition	The power system configuration *N* defines the number of usable power supplies in the power system.
	Input voltage rail	An input voltage rail between 20V and 30V is provided to the power system.
XP_PWR_GEN_003	Output voltage rails	The power system shall provide at least 4 independent output voltage rails.
XP_PWR_GEN_004	Power interface definition	A power interface is a set of two power terminals providing the positive and negative polarities of a given voltage rail to other subsystems.
XP_PWR_PD_005	Input power interface	The power system shall expose at least two input power interfaces.
XP_PWR_PD_007	Output power interface	The power system shall expose at least two output power interfaces per voltage rail.
XP_PWR_GEN_005	Power capability	The power capability for each power interface shall be proportional to the number of power supplies for N≤4.

1.2.2. Power supply requirements

ID	Title	Description
XP_PWR_GEN_006	Mechanical	Each power supply shall fit in a 130x130x85 bounding box.
XP_PWR_GEN_007	Thermal	Each power supply shall dissipate less than 30W in thermal power.
XP_PWR_PD_002	High-voltage rail	The power system shall supply any voltage between 0V and 60V for at least two voltage rails, called HVA and HVB.
XP_PWR_PD_003	Low-voltage rail	The power system shall supply any voltage between 0V and 20V for at least two voltage rails, called LVA and LVB.
XP_PWR_PD_09	Redundancy	All XP_PWR requirements except XP_PWR_006 and XP_PWR_008 shall be fulfilled even if *N-1* among *N* power supplies fail.
XP_PWR_PD_010	Scalability	All *N* power supplies shall share their load current to have less than 1A difference in current provision on each power supply for each voltage rail.

XP_PWR_PD_011	Robustness	A power supply's MTBF shall be at least 72h.
XP_PWR_PD_012	Power supply current capability	For each power supply, each voltage rail shall provide at least up to 10A of continuous current.
XP_PWR_PD_013	Stability margin	For each power supply, each voltage rail shall have a control system phase margin of at least 60° by design.
XP_PWR_PD_014	Bandwidth	For each power supply, each voltage rail shall have a control system bandwidth of at least 10kHz by design.
XP_PWR_PD_015	Fast transient response	For each power supply, each voltage rail shall have a transient response to a current slew rate of 10A/us bounded to 20% of the nominal voltage between 0ms and 1ms after the transient.
XP_PWR_PD_016	Slow transient response	For each power supply, each voltage rail shall have a transient response to a current slew rate of 10A/us bounded to 5% of the nominal voltage between 1ms and 10ms after the transient.
XP_PWR_PD_017	AC stability	For each power supply, each voltage rail shall have an RMS AC component smaller than 2% of the nominal voltage.

1.2.3. Monitoring requirements

ID	Title	Description
XP_PWR_MON_001	Voltage rails	The power system shall supply at least 4 independent voltage rails.
XP_PWR_MON_002	High-voltage rail	The power system shall supply any voltage between 0V and 48V for at least two voltage rails, called HVA and HVB.
XP_PWR_MON_003	Low-voltage rail	The power system shall supply any voltage between 0V and the system's input voltage for at least two voltage rails, called LVA and LVB.

1.2.4. Safety requirements

ID	Title	Description
XP_PWR_SAF_001	Voltage rails	The power system shall supply at least 4 independent voltage rails.
XP_PWR_SAF_002	High-voltage rail	The power system shall supply any voltage between 0V and 48V for at least two voltage rails, called HVA and HVB.
XP_PWR_SAF_003	Low-voltage rail	The power system shall supply any voltage between 0V and the system's input voltage for at least two voltage rails, called LVA and LVB.

1.2.5. Interface requirements

ID	Title	Description
XP_PWR_IF_001	Wireless interface	The power system shall provide a wireless interface compliant with the TBD interface specification.
XP_PWR_IF_002	CAN interface	The power system shall provide a CAN bus interface compliant with the TBD interface specification.
XP_PWR_IF_003	Interactive user interface	The power system shall provide an interactive interface compliant with the TBD interface specification.

2. General architecture

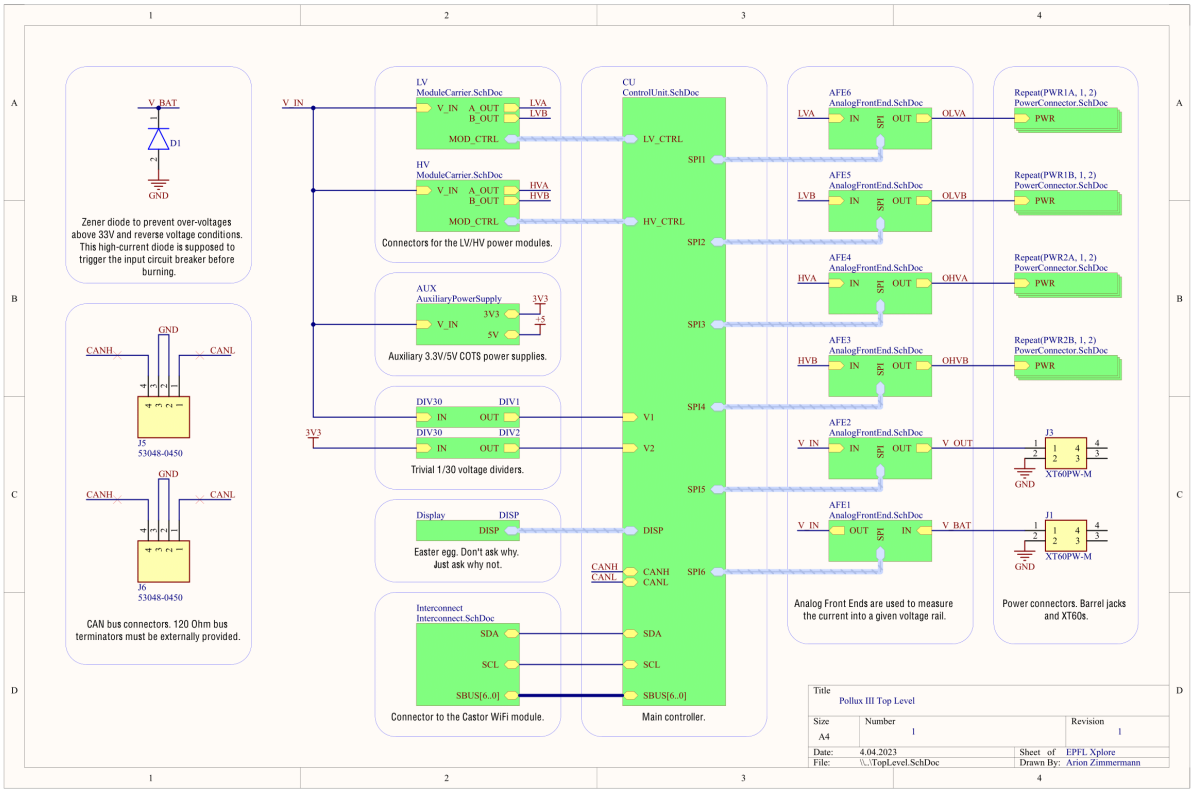
3. Hardware

3.1. Pollux III

Pollux III is the power system's motherboard. It provides a current path between the power modules and the power interfaces, while actively monitoring the current and voltages on the lines. The mission data gathered by the monitors is stored in a persistent storage and transmitted to Castor, to a CAN bus and to a touchscreen.

3.1.1. Top-level

Pollux III's top-level schematic is depicted in @. It represents the block diagram and high-level connections between the different components that compose Pollux.



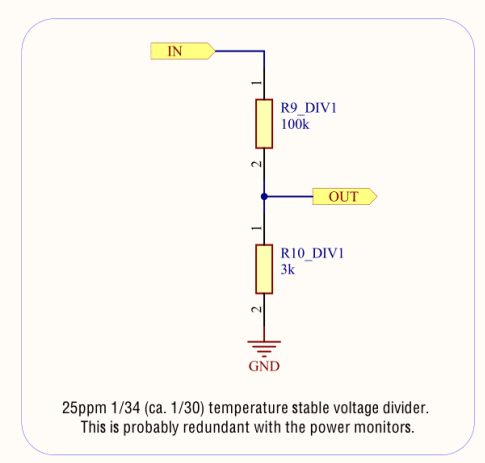
Pollux is a motherboard which accepts up to four power modules. The latter are the ones responsible for converting the battery's voltage to the required voltages. Four of them are defined:

- **LVA**: Low-voltage power module channel A.
- **LVB**: Low-voltage power module channel B.
- **HVA**: High-voltage power module channel A.
- **HVB**: High-voltage power module channel B.

In addition to this, two additional power interfaces are defined:

- **Input bus**: The power input coming from the safety circuitry.
- **Follower bus**: The power output that has the same voltage as the power input.

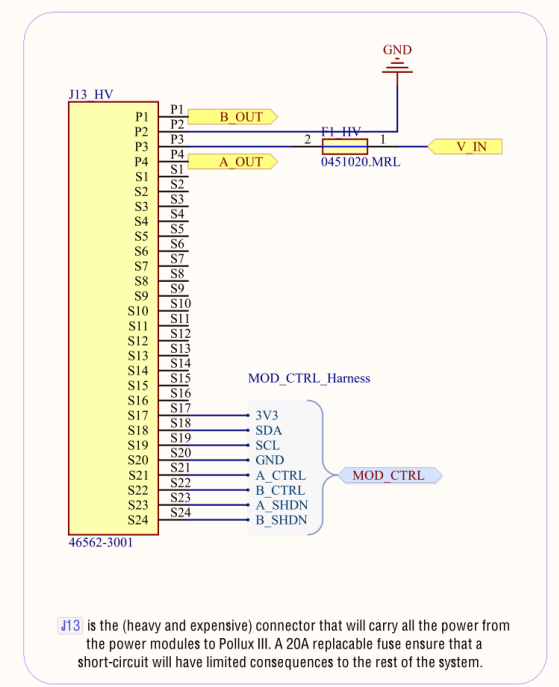
XT-60 power connectors are used for the input and follower busses, as they are designed to carry more current. As a safety measure, a high-power Zener diode is placed in parallel to these power buses, to prevent damage that would be caused by a reverse-polarity or over-voltage condition.
DC barrel jack rated for 10A continuous current are used for the four voltage rail outputs.
Six measuring devices, called the Analog Front Ends, are used to measure the voltage and current at the power interfaces. These measuring devices all have a digital communication bus connected to the main control unit.
Moreover, analog inputs to Pollux' main control unit are used to sense the battery voltage rail and the 3.3V voltage rail. A high-precision temperature-stabilized resistor divider, shown in schematic @, is used to measure the output voltage relatively accurately.

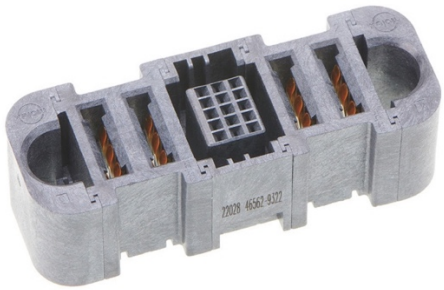


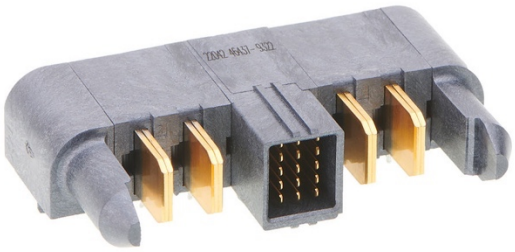
Finally, digital interfaces, such as the Castor interface, the CAN bus interface and the Interactive User Interface, are also connected to the main control unit.

3.1.2. Power Interface

Schematic @ shows the pin specification of the power interface between the power modules and the Pollux motherboard. The mechanical stiffness of the power connector assembly after mating was the major weakness of Pollux II. Instead, a Molex EXTreme Ten60Power heavy-duty connector is employed in Pollux III, significantly reducing the stiffness issue of Pollux II.







A 20A fuse on the input side of the power module prevents damage to the power system if one of the power modules fails in short-circuit.
Table @ lists and describes the pin interfaces of the power modules.

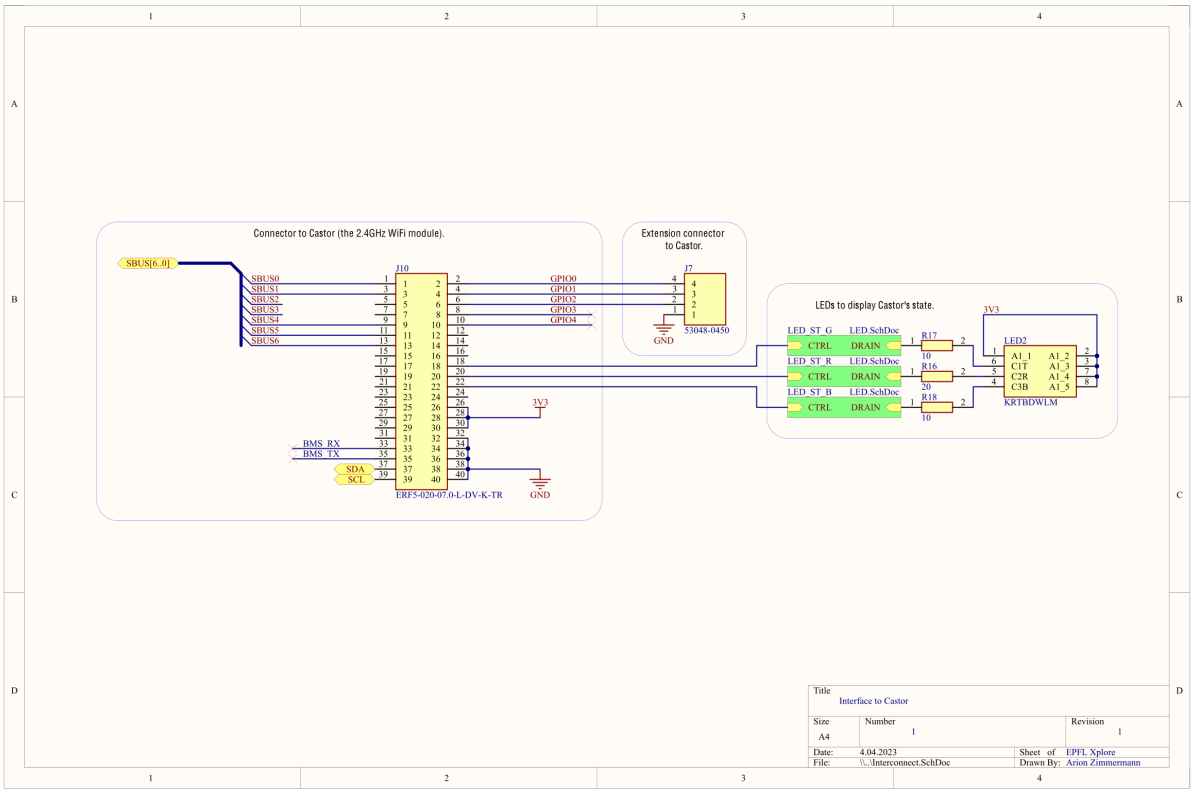
Pin	**Corresponding function**
P1 (max. 60A)	Channel B voltage output
P2 (max. 60A)	Power ground

P3 (max. 60A)	Fuse-protected input from 20V to 30V
P4 (max. 60A)	Channel A voltage output
S17	3.3V input voltage reference
S18	I2C data line SDA to control the output voltages
S19	I2C clock line SCL to control the output voltages
S20	Digital ground reference
S21	Channel A Enable Pin (active-high, pulled-up)
S22	Channel B Enable Pin (active-high, pulled-up)
S23	Channel A Shutdown Pin (active-high, pulled-down)
S24	Channel A Shutdown Pin (active-high, pulled-down)

3.1.3. Castor Interface

A physical separation exists between the power system's motherboard and the wireless control module. Those two boards are interconnected through a 40-pins connector that allows signals to be transmitted from one board to another. The pin assignment for this connector is very generic to allow forward-compatibility and is described in Appendix A. This bus consists of several subinterfaces: a sensor bus, a system bus, a general purpose bus and a status bus.

Schematic @ corresponds to this interface and is shown below for a better understanding.



3.1.3.1. Sensor bus interface definition

The link between the wireless control board and the motherboard is implemented through the UART protocol. Up to two UART ports can be used for transmission. This dual port feature was used with Pollux II but is deprecated in Pollux III, where only the UART port 1 is significant.

Pin	**Corresponding function**
Sensor bus signal 0 (SBUS0)	Transmit on UART TX port 1

Sensor bus signal 1 (SBUS1)	Receive on UART RX port 1
Sensor bus signal 2 (SBUS2)	*Unused (legacy)*
Sensor bus signal 3 (SBUS3)	*Unused (legacy)*
Sensor bus signal 4 (SBUS4)	Serial Audio Interface Frame Sync
Sensor bus signal 5 (SBUS5)	Serial Audio Interface Clock
Sensor bus signal 6 (SBUS6)	Serial Audio Interface Data

3.1.3.2. System bus interface

This interface is used to connect the power supply to the rest of the Rover. In particular, it consists of an UART link to connect to the Rover's BMS (battery management system) and an I^2C connection to the power system's sensors. Once again, the pin assignments are stated below.

Pin	**Corresponding function**
System bus signal 0 (SYSIO0)	UART RX
System bus signal 1 (SYSIO1)	UART TX
System bus signal 2 (SYSIO2)	I^2C SDA
System bus signal 3 (SYSIO3)	I^2C SCL

In Pollux III, the connection to the BMS is not implemented.

3.1.3.3. General purpose bus interface

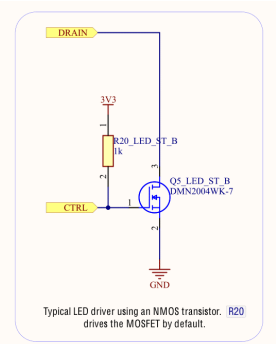
Five general purpose lines also connect the control module to the power module. Those are mainly used for debugging purposes. No special protocol is defined for those general-purpose signals. A 1.25mm Molex PicoBlade header is used to interface the SYSIO0 and SYSIO1 to the BMS.

3.1.3.4. Status bus interface

To indicate the state of the control module, three signals connect the control module to the power module. These lines shall carry PWM signals whose duty cycle completely defines the state of the control module.

The PWM signals are used to drive a NMOS MOSFET as in schematics @. When the control pin (CTRL) is floating (Castor reset or disconnected) the MOSFET conducts and the RGB LED shines white.

When the control signal is a PWM, the MOSFET drain's voltage follows it and the LED shines accordingly. The duty cycle of the three PWM signals change the intensity of the three RGB color channels.

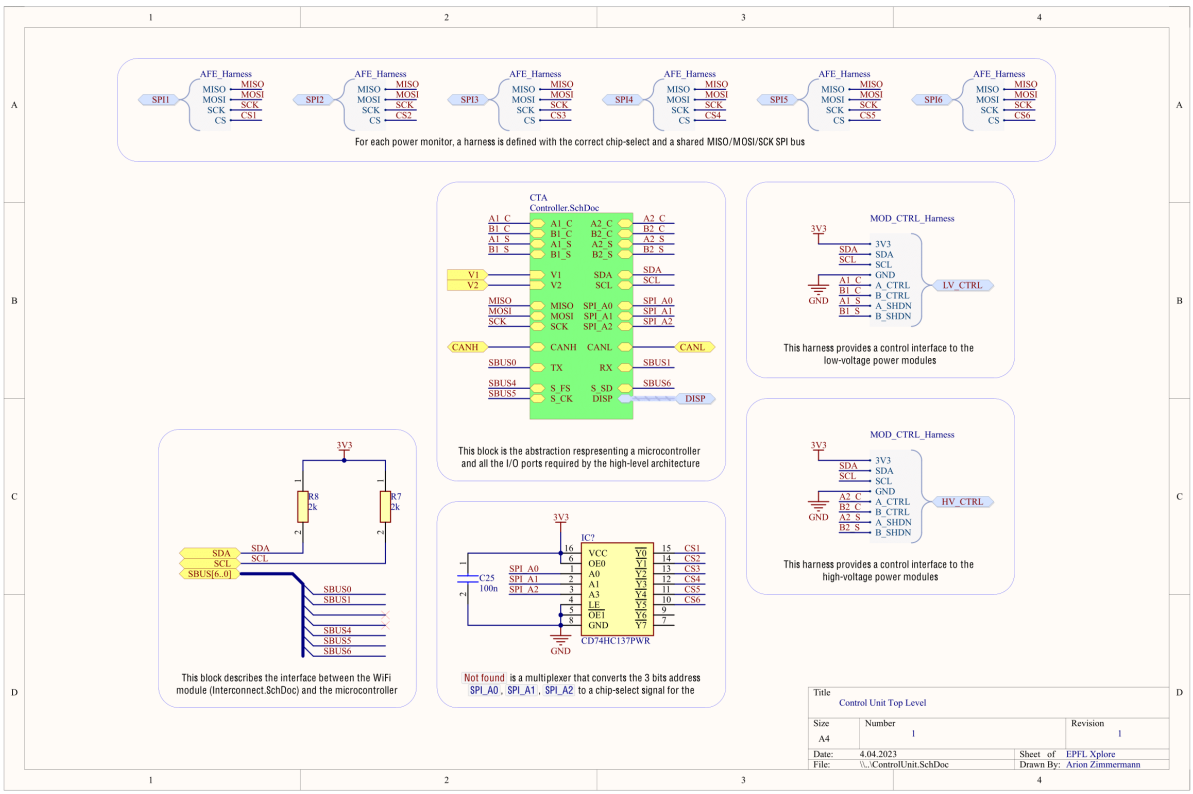


3.1.3.5. Power interface

A 3.3V bi-directional power port connects Castor to Pollux. One the one hand, if Castor is powered by USB, Pollux also receives a 3.3V input supply. On the other hand, if Pollux is powered by the battery, Castor receives a 3.3V input. The current specification for the power port is 2A maximum.

3.1.4. Main controller

The control unit's schematic @ describes how the main controller is interfaced with the multiple sensors, power modules, and external interfaces.



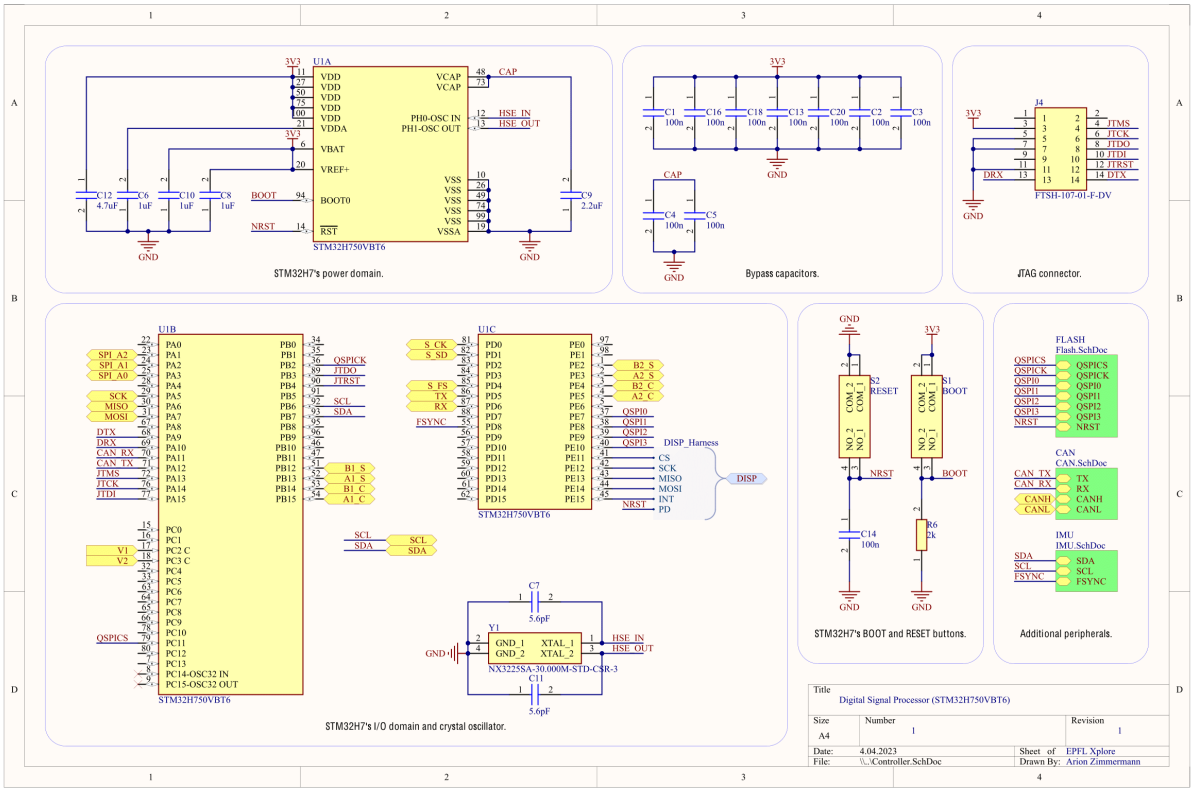
A SPI communication bus allows the main controller to communicate with the analog front ends, which measure the voltage and current through all power interfaces.
In the SPI protocol, a controller can only communicate to one sensor at a time. The active sensor is selected through an active-low chip select (nCS). With this regard, an 8-bits active-low multiplexer CD74HC137 is used to convert a 3-bits address, which defines the sensor being selected, to an 8-bits signal, which drives the nCS pins of the analog front ends.

Table @ lists the maps the address values to the selected sensor.

Address	Chip-select port	Selected analog front end
000	11111110	Low-voltage channel A (LVA) bus
001	11111101	Low-voltage channel B (LVB) bus
010	11111011	High-voltage channel A (HVA) bus
011	11110111	High-voltage channel B (HVB) bus
100	11101111	Follower bus
101	11011111	Input bus
110	10111111	*None*
111	01111111	*None*

The harness *MOD_CTRL_Harness* encompasses all the necessary signals to fully control the state of a power supply. An I2C bus allows the main control unit to communicate with the supply to change its output voltages. By design, the power modules I2C addresses should be

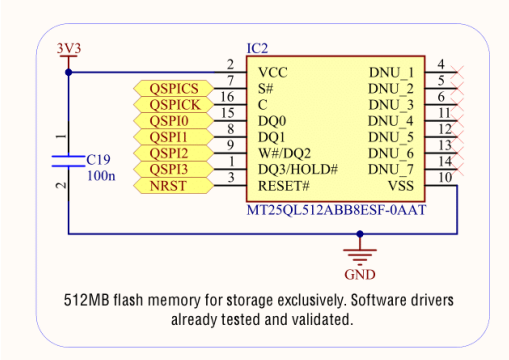
unique, so that the main controller can address and communicate with them separately, even though they are all connected to the same I2C bus. Two 2kΩ pull-up resistors are added to the SDA and SCL lines as per the I2C bus specification. A low resistor value ensures that the Fast-Mode I2C communication can be performed, at the cost of a higher power dissipation. Additionally, four digital lines control the state of the supplies and are connected separately to the main controller.
The controller itself in defined in schematic @. It uses an STM32H750VBT6 microcontroller from STMicroelectronics as the main controller. In the schematic, the STM32H750 microcontroller is divided into three parts for better readability.



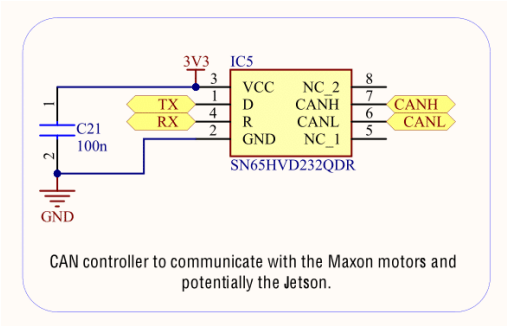
A 30MHz crystal oscillator from NDK is used as the microcontroller's High-Speed External Oscillator (HSE-OSC). This oscillator presents a load capacitance of 8pF. To compute the value of the C7 and C11 capacitors, the estimated stray capacitance of the link between the oscillator and the STM32 microcontroller must be taken into account. After a preliminary layout of the PCB, an estimated stray capacitance of 5.2pF is present on the line. According to the application note AN2867 from STMicroelectronics, section 3.2, this leads to C7 and C11 capacitor values of 5.6pF each.
The programming interface of the STM32 is exposed as a JTAG connector from Samtec, a reset button and an active-high boot mode selection button. Moreover, a 100nF anti-bounce capacitor C14 is added on the reset signal line.
Note that the bypass ceramic capacitors are placed according to the microcontroller's datasheet.
In addition to this, three peripherals are connected to the microcontroller:

- One 512Mb flash memory to store persistent mission data (Schematic @).

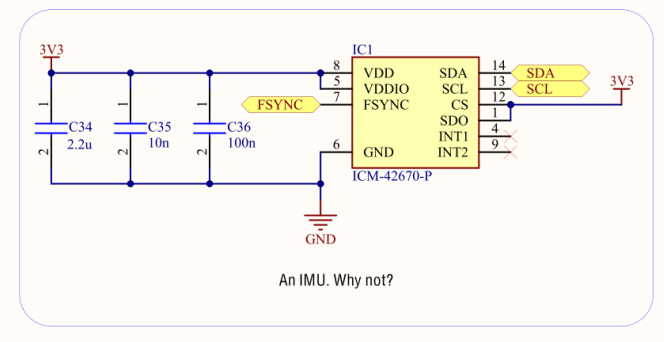
- One CAN controller to communicate with other subsystems in the rover (Schematic @).
- One IMU to compute the attitude of the rover before launching the Brokkoly drone (Schematic @).
-



512MB flash memory for storage exclusively. Software drivers already tested and validated.



CAN controller to communicate with the Maxon motors and potentially the Jetson.



An IMU. Why not?

3.1.5. Analog Front End

The power system specifications require an accurate measurement of output voltage, output current and power consumption for every power interface. To satisfy those requirements, it is necessary to make several design choices.

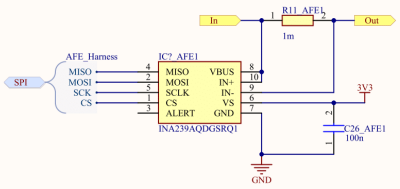
First of all, a high-side current sensing method is used instead of a low-side sensing. This has the advantage of rejecting the disturbances that could occur on the ground plane, as well as giving the operator the ability to detect load short-circuits [RF10].

Moreover, there are two ways of measuring DC currents: Hall sensing and shunt resistor sensing. Hall sensing measures the magnetic field generated by a wire according to Ampere's law, whereas shunt sensing measures the voltage across a small-value resistor. Although Hall sensing is non-intrusive and does not affect the efficiency of the converter, it is more complex to implement and takes more space on the PCB. A shunt resistor sensing method is therefore preferred. Since up to 40A can be expected on the input and follower power ports, a high-quality current sense resistor of 1mΩ is used for an expected power dissipation of 1.6W (below the 2W rating of the resistor). The temperature stability of this resistor is reduced to 50PPM/°C, so that its impact on the current measurements is minimized.

After that, an INA239 power monitor from Texas Instruments communicating through SPI is used to measure the current and voltage through the current sense resistor.

The INA239 IC was specifically chosen because it rejects the common-mode component of the differential signal up to 80V. This is particularly useful since the power modules can deliver a voltage up to 60V according to their specification.

The analog front end system is represented in schematic @.



SPI delta-sigma power monitor with embedded temperature sensor. 16-bits. 160mV full-scale. Maximum sensed current: 40A (shunt thermal limit)

Title	Power Monitor		
Size	Number		Revision
A4	1		1
Date:	4.04.2023	Sheet of	EPFL Xplore
File:	\\..\AnalogFrontEnd.SchDoc	Drawn By:	Arion Zimmermann

3.1.6. Auxiliary Power Supplies

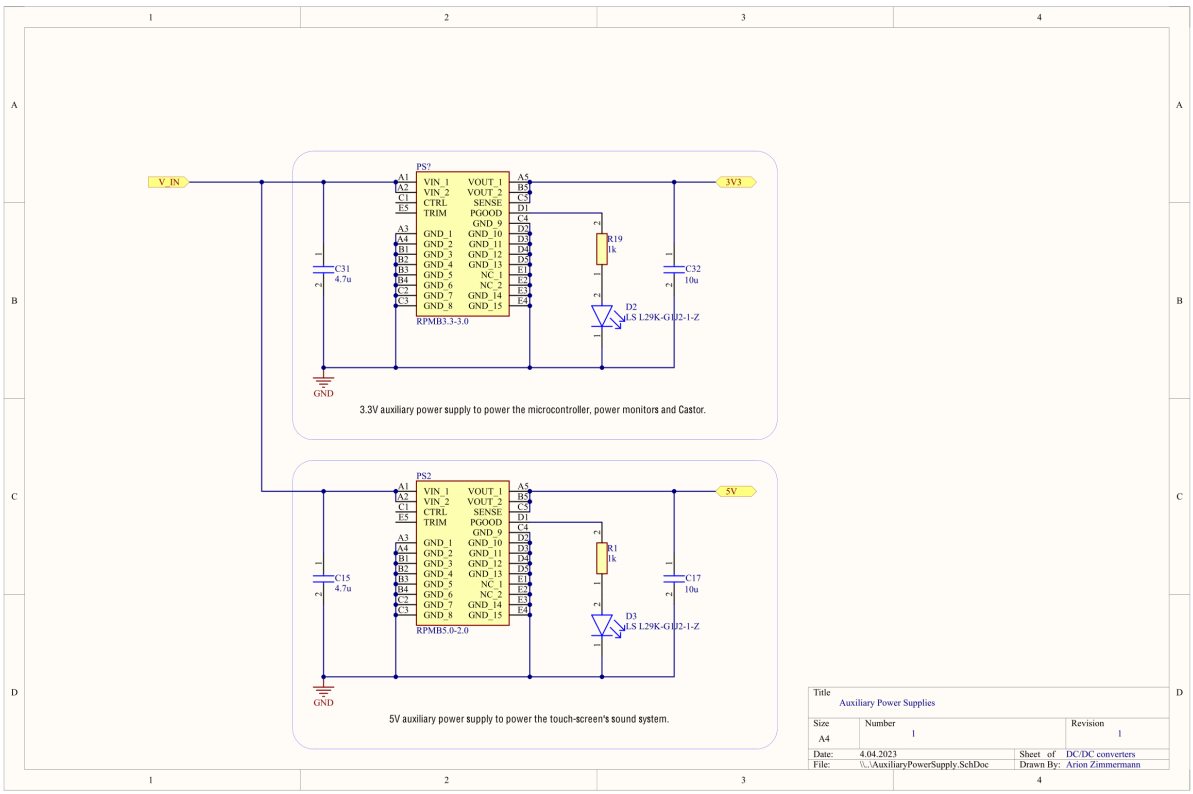
Two auxiliary power supplies are used to deliver a 3.3V rail and a 5V rail to power the multiple ICs present on Pollux. A fully integrated commercial solution was used from RECOM power, to minimize their risk of failure. Two RECOM RPMB power supplies are used, one RPMB3.3-3.0 (3.3V, max. 3A) and one RPMB5.0-2.0 (5V, max. 2A). Two LEDs with a series resistor indicate the state of the power supplies. If the power supplies are “Power Good” by activating the PGOOD signal, the LEDs shine green.

RECOM power supplies were chosen because they are fully integrated and can provide a high output current. The ease of use and simplicity of the datasheet were also considered in the selection. Even though these power supplies are soldered from below, manufacturing mistakes are almost impossible in the soldering process.

Bypass stabilizing capacitors are added on the input and output lines, according to the datasheet.

A bypass input ceramic capacitor is placed at the VIN_1 and VIN_2 pads to ensure that the supply remains regulating at the right output voltage even when voltage drops occur on the input line. Another ceramic capacitor is placed on the output 3V3 line to ensure that the power supply remains stable over all current ranges between 0A and 3A.

@ LED



3.1.7. User Interface

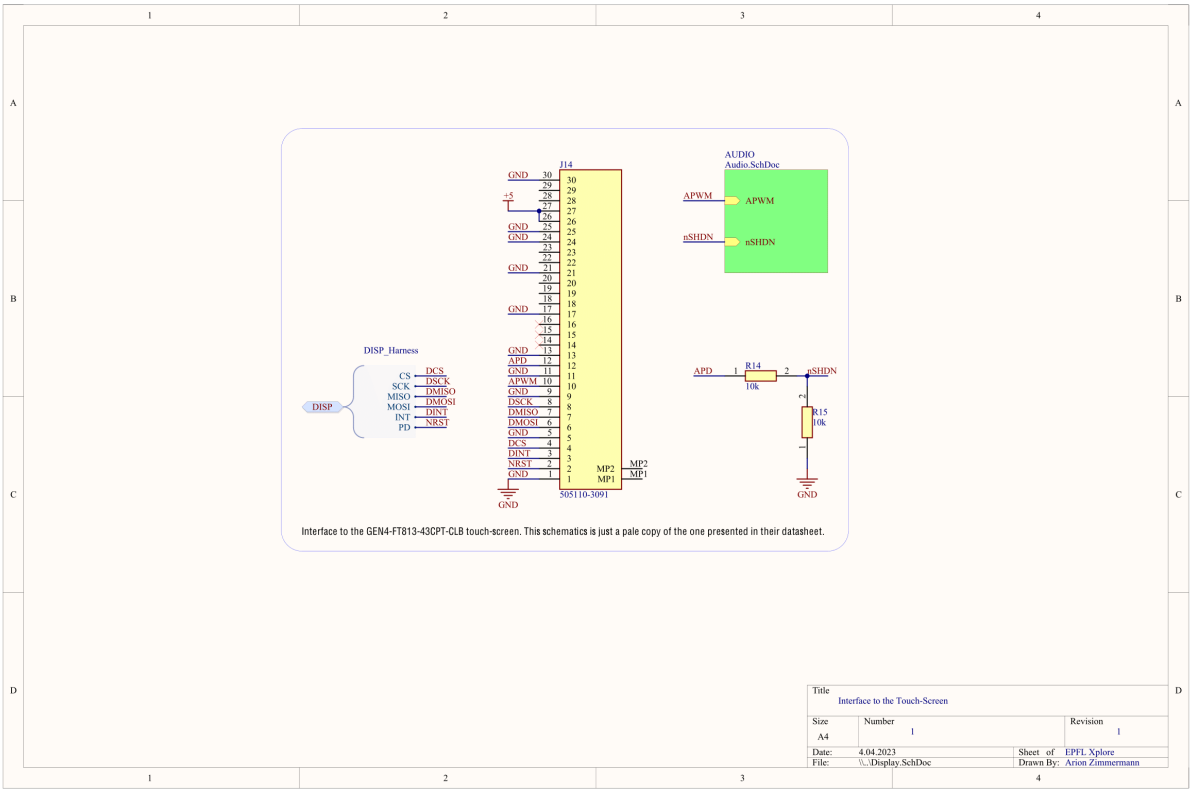
An interactive user interface is implemented in Pollux III. It makes use of one touchscreen and one sound system to give the end-user a direct control and feedback of the power system.

The chosen integrated commercial solution is a GEN4-FT813-43CPT-CLB and combines a touchscreen and an FT813 graphical processor. The processor is controlled by the main controller through a *DISP_Harness*, which mainly incorporates an SPI communication bus. The integrated solution proposed by GEN4 was selected because it incorporates a well-documented FT813 IC, a high-current backlighting system to ensure that the screen is visible in full daylight and a high-resolution capacitive touchscreen.

The FT813 controller also has the ability to control a sound system through a PWM audio signal. Schematic @ depicts the interactive user interface implementation.

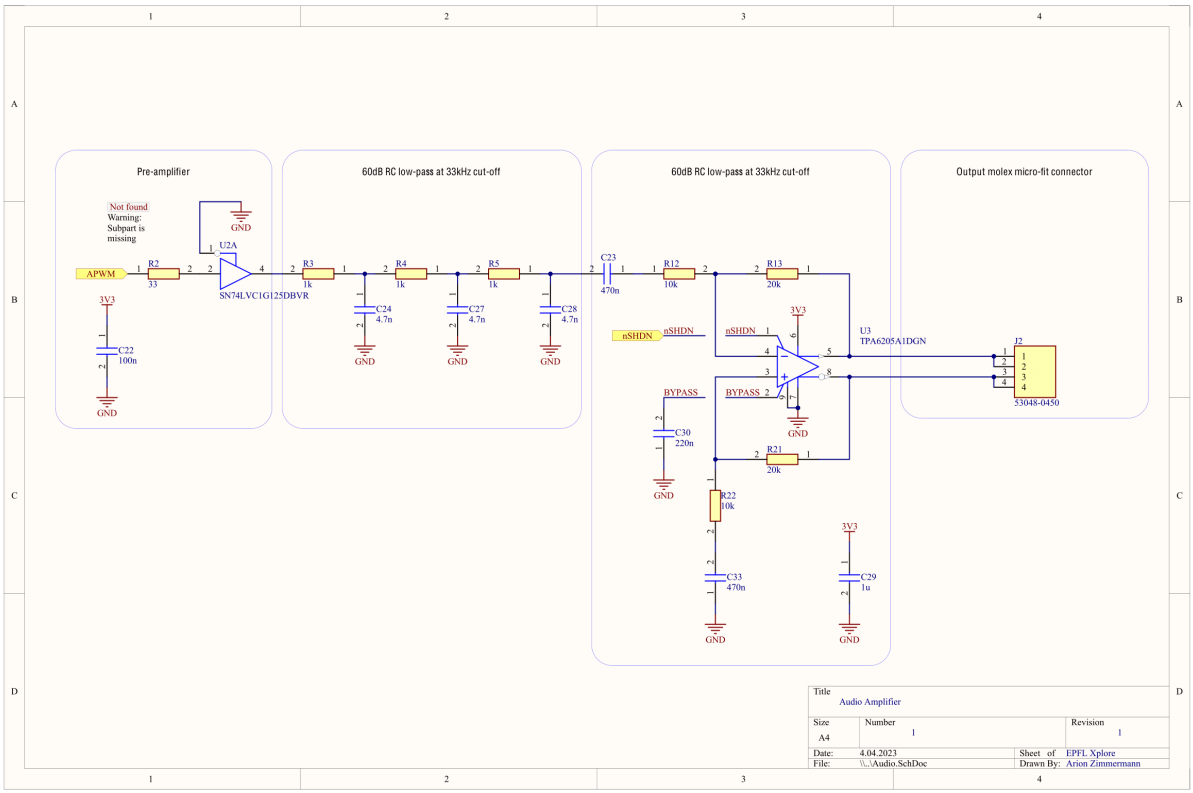
A 50% voltage divider is applied to the Audio power-down pin, as per the datasheet specification.

The touchscreen embedded in the commercial solution is a 4.3’’ capacitive touchscreen with bezels. It fits exactly the form-factor of the final Pollux PCB.



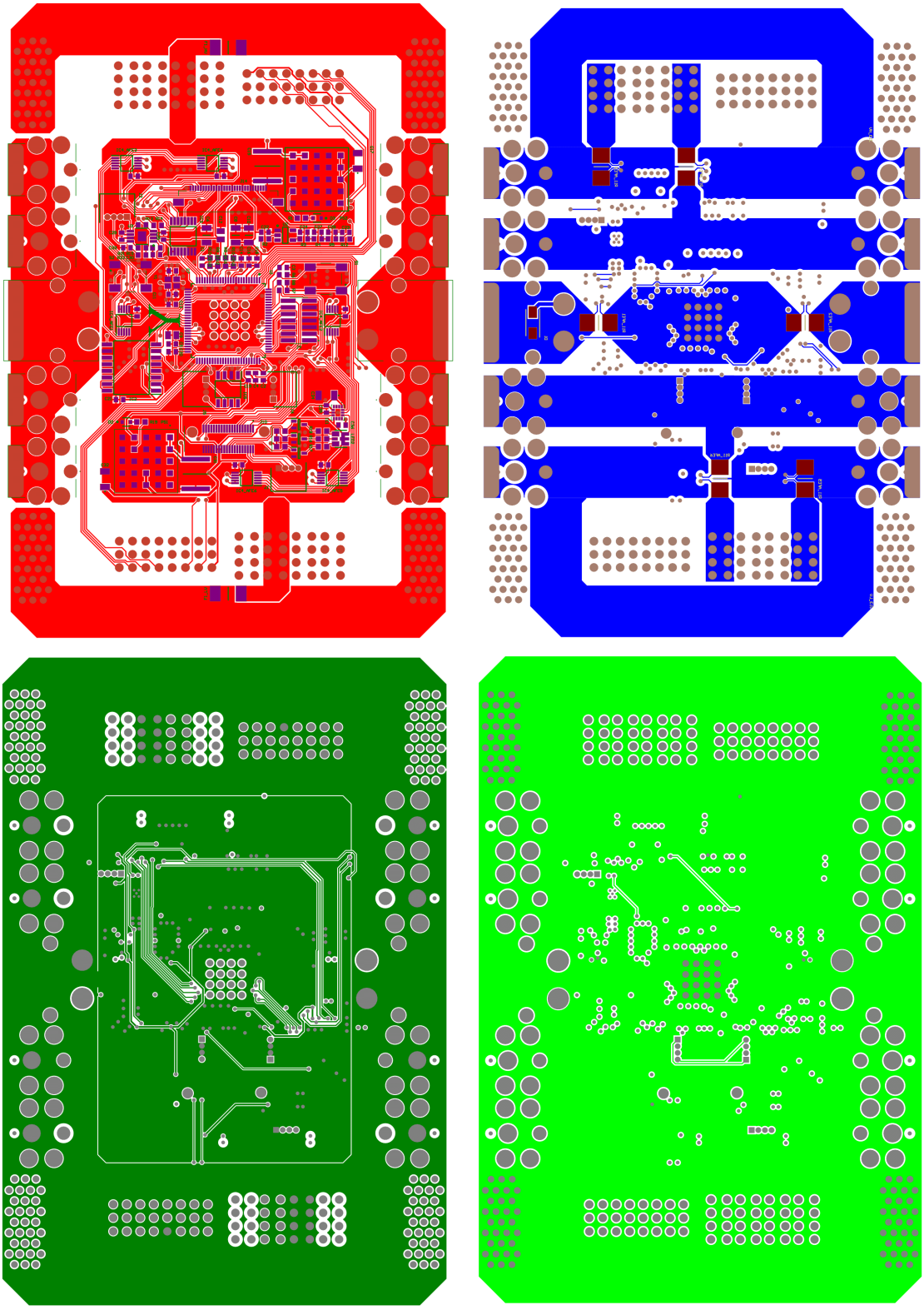
In schematic @, the sound system is abstracted by the Audio block which is detailed in further details in schematic @.

The sound system schematic in fully inspired by the advised design in the GEN4-FT813 datasheet. It makes use of a pre-amplifier, a simple 60dB 33kHz RC low-pass filter and a push-pull differential amplifier. The output audio signal is exposed on Pollux by a Molex MicroFit connector.



3.1.8. Layout

The Pollux III final PCB is drawn in layout @. It consists of four layers, two (top and bottom) for placing the components and high-current power paths, one *groundish* layer and one *powerish* layer. The suffix *-ish* is added to the name of the layers to denote that they are sometimes used for other purposes than the one implied by the layer name.



The difficuty in designing the Pollux III motherboard resides in having to carry large amounts of current through the PCB, without impacting the control and measurement systems. This is particularly challenging when dealing with very compact PCBs.

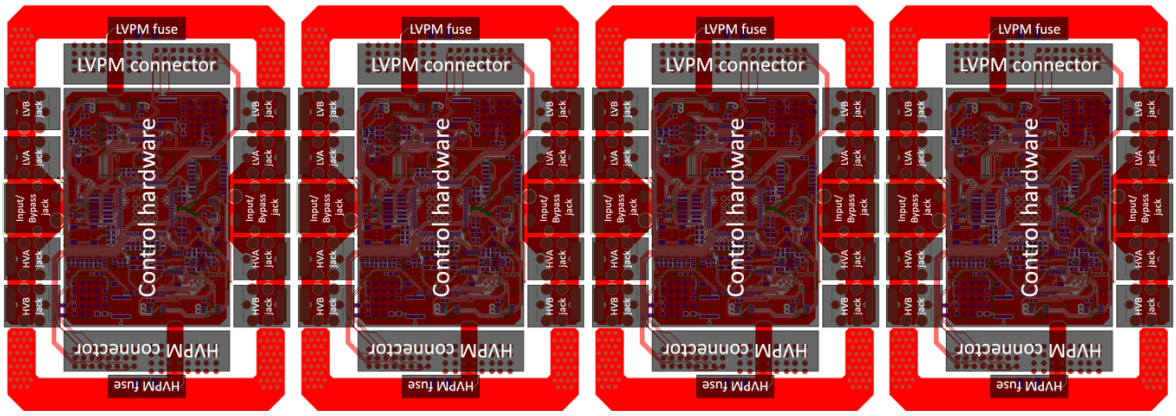
The very first design consideration here is to fulfil the system requirement:

XP_PWR_GEN_001: The power system consists of up to N power supplies.

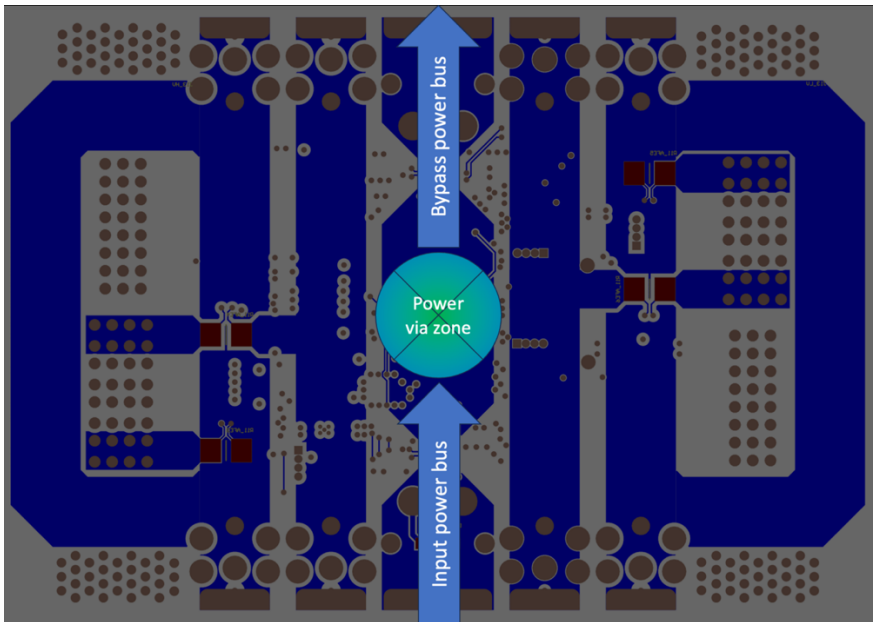
In other words, the systems engineering requires the power system to be fully scalable. The easiest solution for this is to design Pollux III so that it is mechanically stackable with other motherboards. Diagram @ shows how a 4-power supplies configuration could look like.

Special solder pads on the top and on the bottom layer can be used to connect two Pollux III motherboards with one another. These solder pads are also compatible with the barrel jack footprints, allowing the two extremities of the power system to interface with the subsystems.

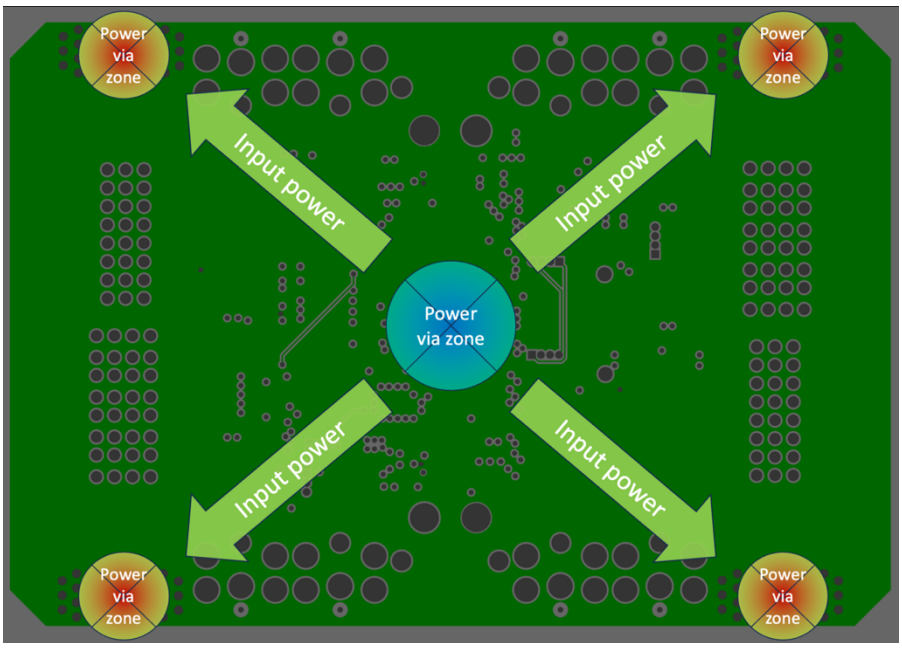
Throughout the design, a sense of central symmetry was preponderant, so that each Pollux III motherboard could be completely rotated around its center and still be interfaceable with other motherboards. Only the logical names LVA and LVB would have to be swapped to HVA and HVB respectively. Note that the connector order LVB – LVA – Input/Bypass – HVA – HVB is also a direct consequence of this central symmetry.



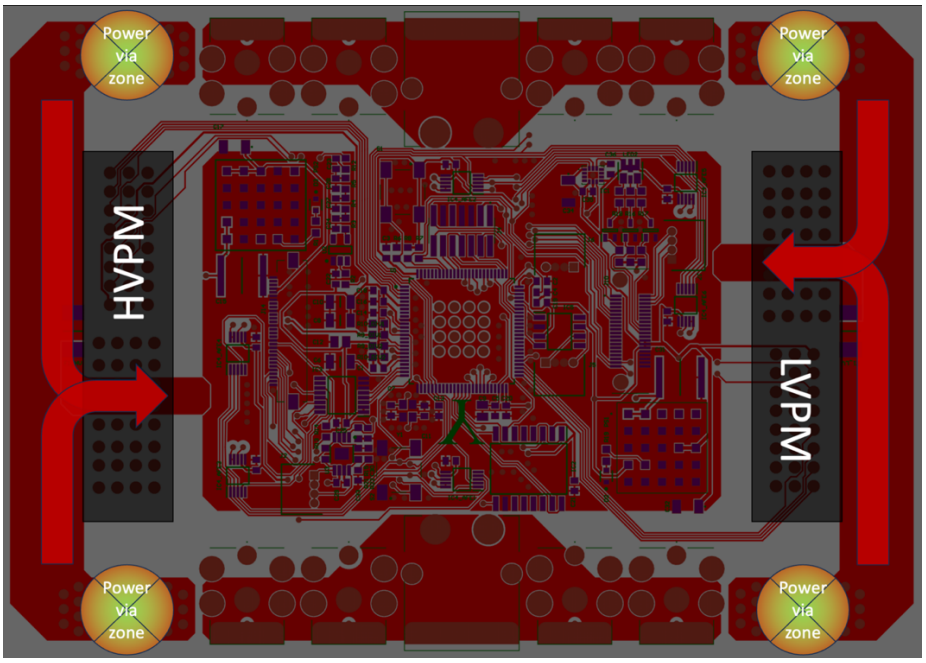
Once this motherboard stacking architecture is adopted, it was necessary to design the high-current power path between the Input/Bypass ports to the subsystems. Diagram @ depicts how the input power is passed to the bypass output bus after being tapped by a high-power via zone consiting of 16 vias, each capable of carrying up to 3.5A.



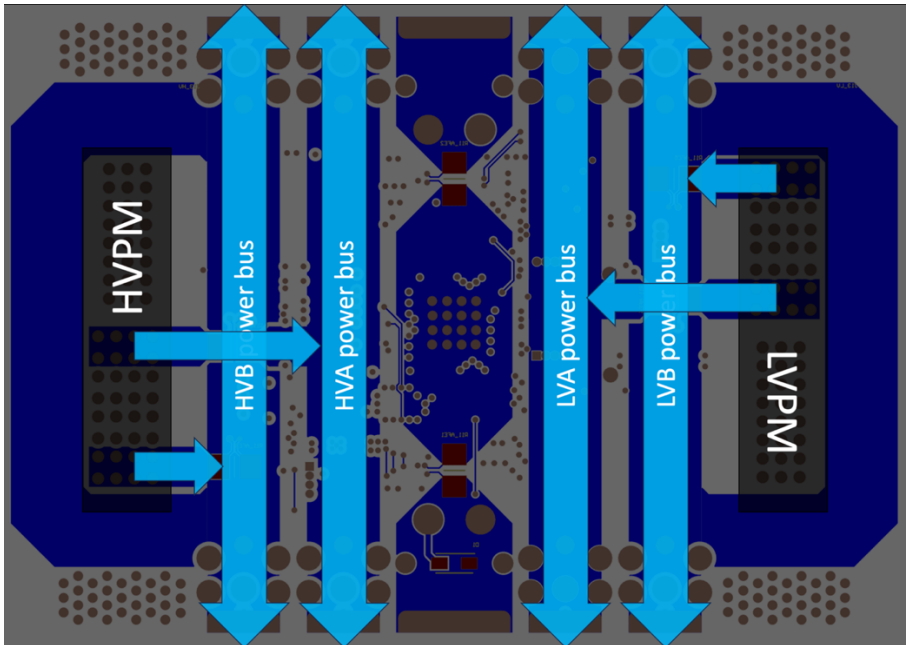
This via tap sources the powerish layer. At this point the powerish layer has a voltage roughly equal to the (battery) input voltage. Four other via taps bring this voltage to the extremities of the PCB, as shown in diagram @.



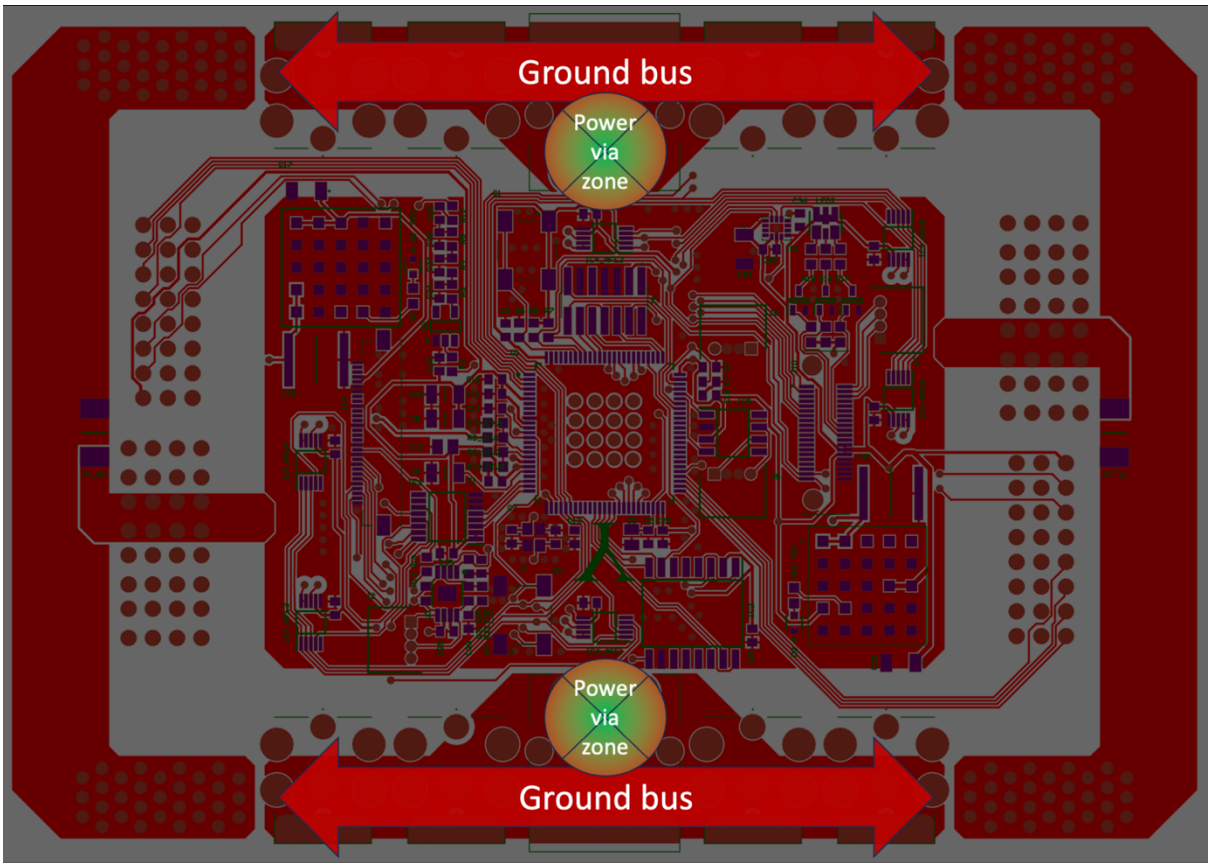
Large tracks are finally used to bring the input voltage to the power modules on the top layer as in diagram @.



The power modules convert the input voltage to voltages required by the subsystems and feeds them back to the power busses through large tracks in the bottom layer (diagram @).

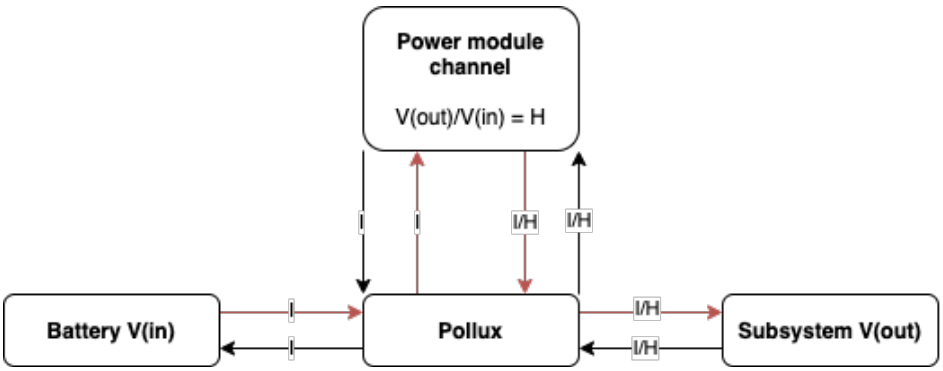


Large tracks finally connect the subsystems grounds together to the battery voltage in diagram @.



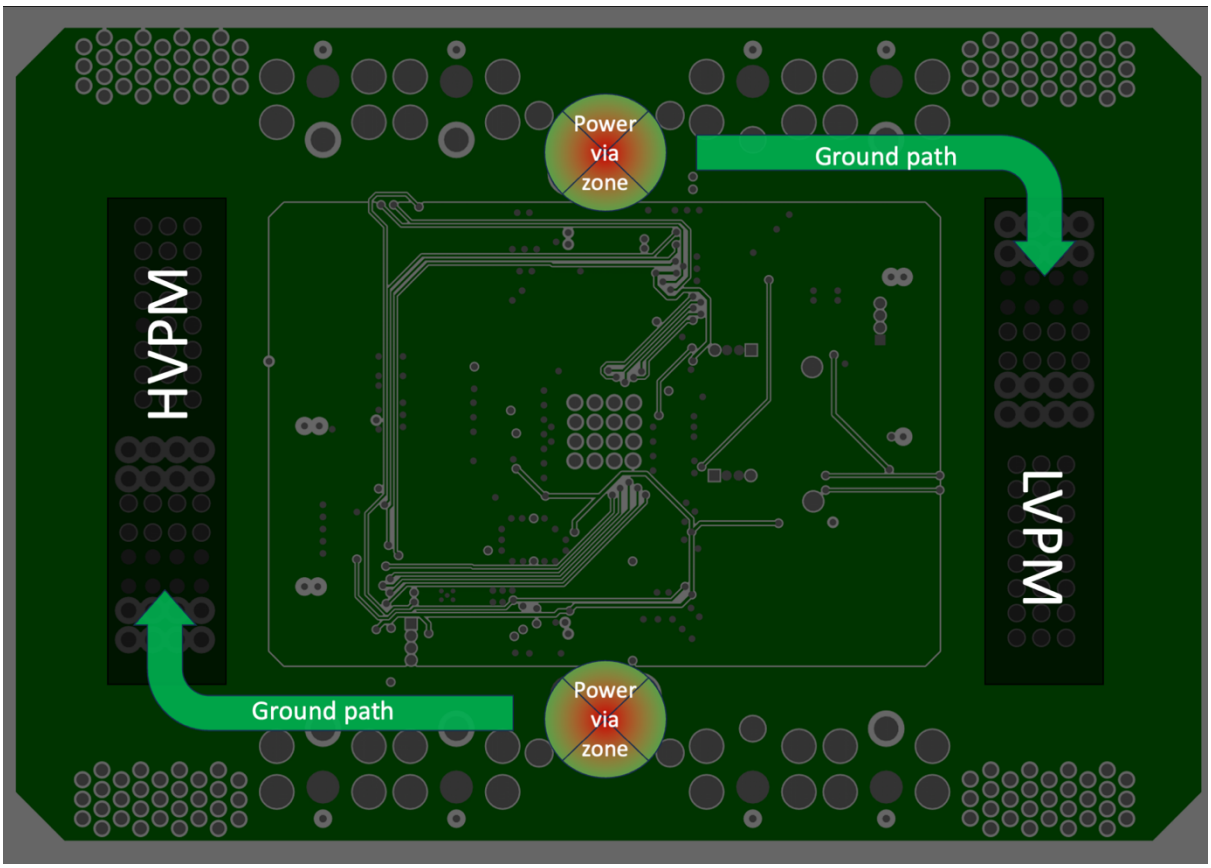
One might be tempted to think that the ground current of the subsystem is exactly equal to the battery ground current and that all currents get cancelled out in the so-called ground bus of diagram @. However, this intuition is incorrect because the voltage levels at the subsystem outputs differ from the input voltage level.

Figure @ shows how the different currents must be summed up (red are voltage levels and black are grounds; I is the current output from the battery and H is the voltage transfer function of the power module). Indeed, the total ground current from the power modules is *I – I/H*. Contrary to intuition, this ground current can become very large, especially when buck converters are used to convert the input voltage to a low voltage, such as 5V (H=0.2).

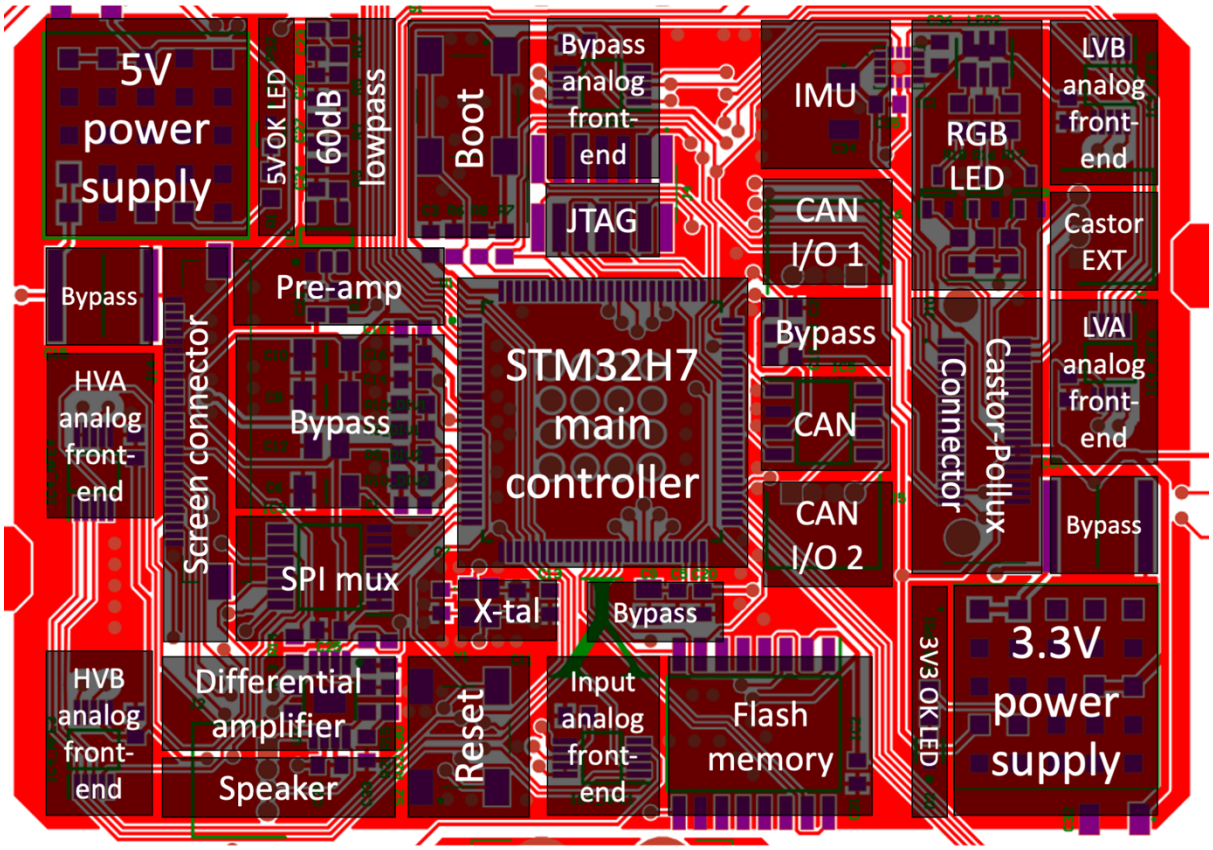


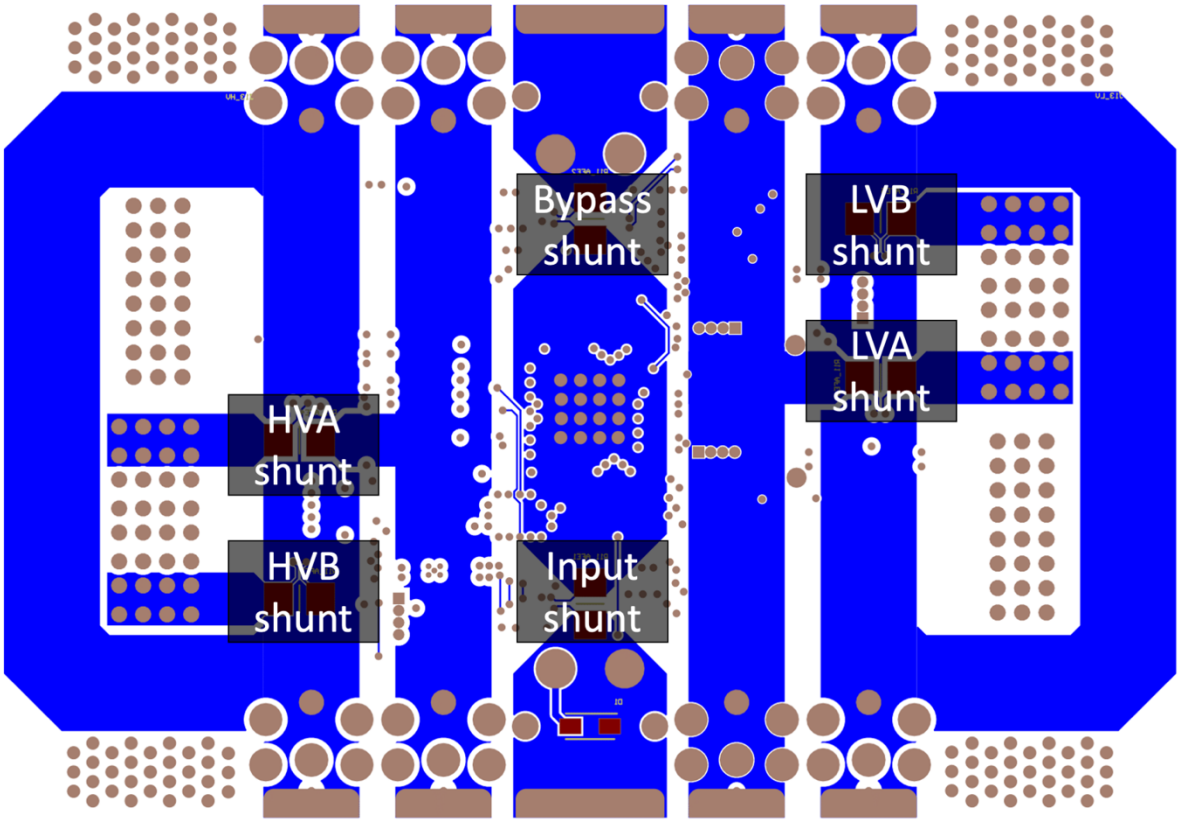
This consideration highlights the importance of using a via tap to the groundish layer, where a ground current can be brought to the power modules. To minimize the ground

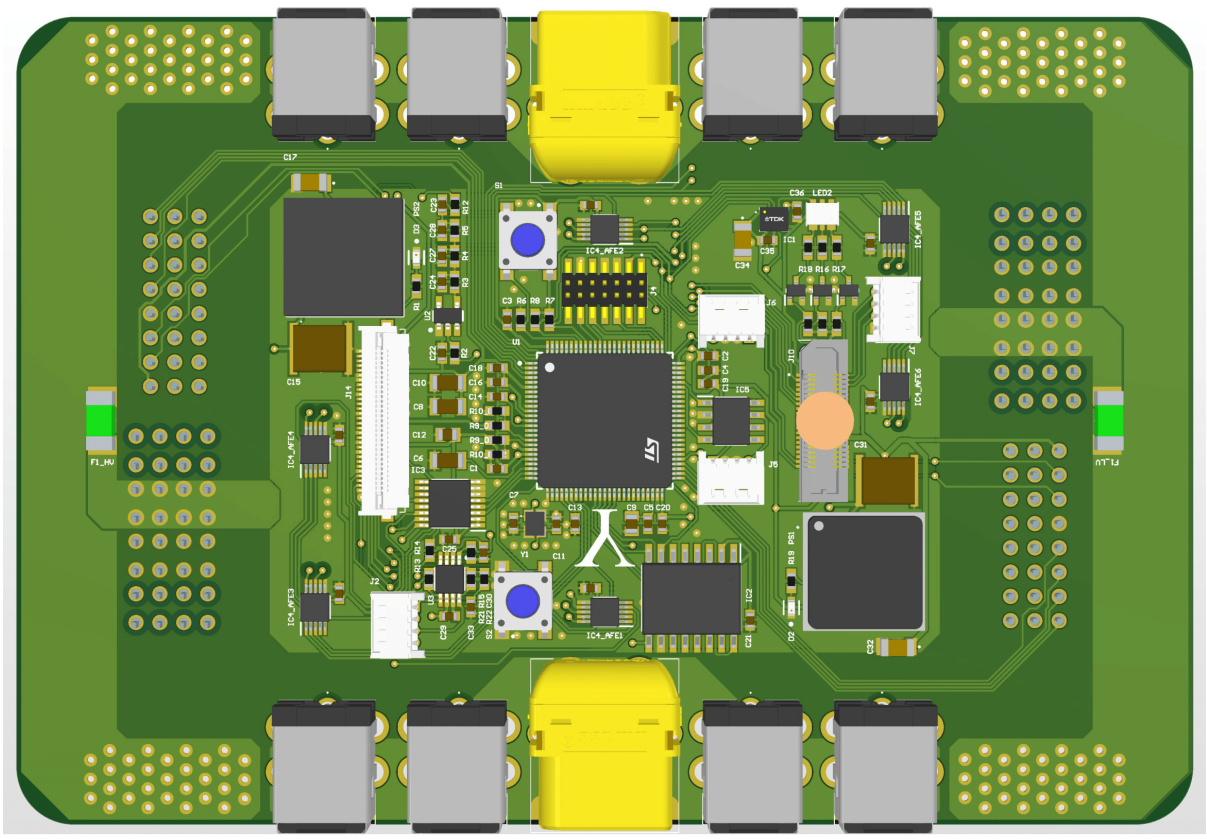
noise due to this, a 0.5mm barrier isolates the control/measurement ground from the power ground. A single 5mm connection connects the power ground to the control/measurement ground.

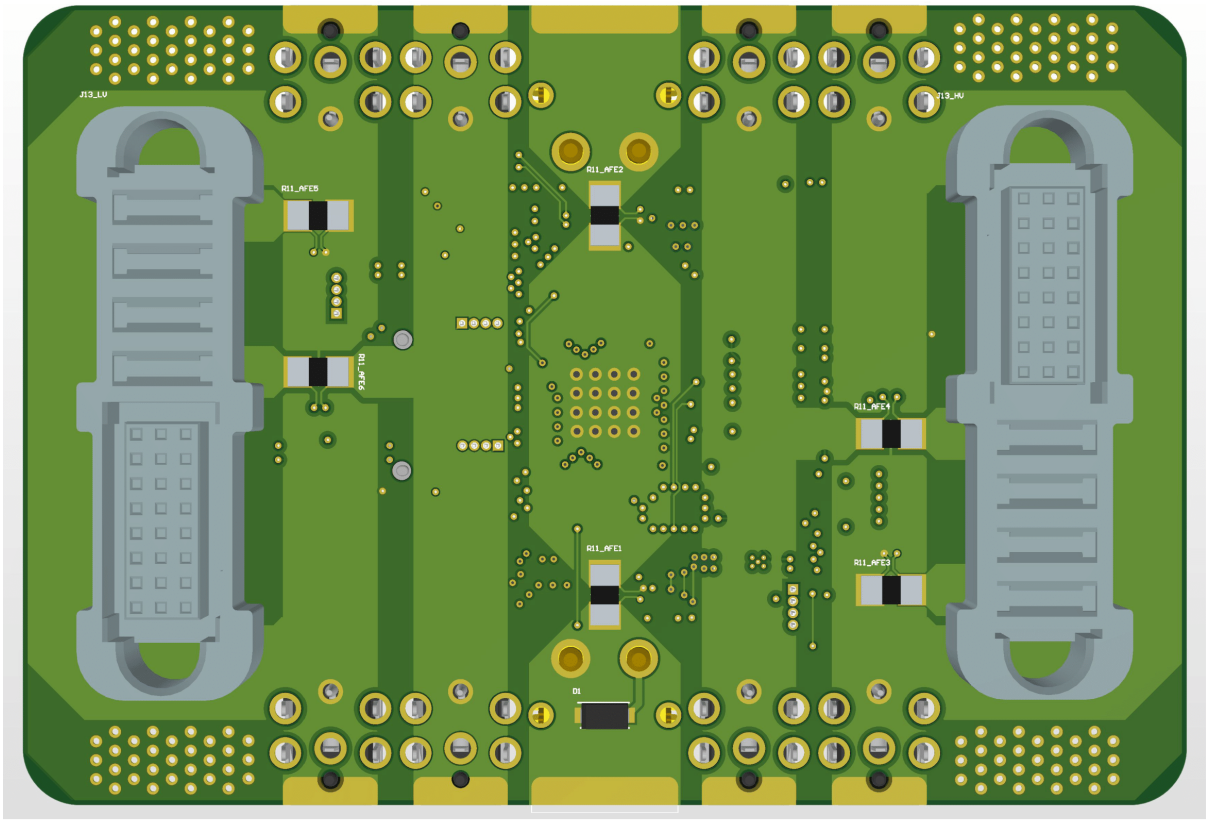


The next step after deciding which is the power path is to place the different components on the top and bottom layers. For simplicity, most components were installed on the top layer since it greatly simplifies the soldering of the PCB. Only shunt resistors were placed on the bottom layer, to measure the power consumption on each voltage rails.


5V power supply
5V OK LED
60dB lowpass
Boot
Bypass analog front-end
JTAG
IMU
RGB LED
LVB analog front-end
CAN I/O 1
Castor EXT
Bypass
Pre-amp
HVA analog front-end
Screen connector
Bypass
STM32H7 main controller
Bypass
CAN
Castor-Pollux Connector
LVA analog front-end
CAN I/O 2
Bypass
SPI mux
X-tal
Bypass
3V3 OK LED
3.3V power supply
HVB analog front-end
Differential amplifier
Reset
Input analog front-end
Flash memory
Speaker


Bypass shunt
LVB shunt
LVA shunt
HVA shunt
HVB shunt
Input shunt





3.2. Power module design

3.2.1. Topologies

3.2.2. Efficiency

3.2.3. Stability

3.2.4. Thermal management

3.3. Power modules common design

The power modules compatible with Pollux III make use of a common power interface, as described in section 3.1.3.5.

3.3.1. Interface

Table @ (repeated from table @) lists and describes the pin interfaces of the power modules.

Pin	**Corresponding function**
P1 (max. 60A)	Channel B voltage output
P2 (max. 60A)	Power ground
P3 (max. 60A)	Fuse-protected input from 20V to 30V
P4 (max. 60A)	Channel A voltage output
S17	3.3V input voltage reference
S18	I2C data line SDA to control the output voltages
S19	I2C clock line SCL to control the output voltages
S20	Digital ground reference
S21	Channel A Enable Pin (active-high, pulled-up)
S22	Channel B Enable Pin (active-high, pulled-up)
S23	Channel A Shutdown Pin (active-high, pulled-down)
S24	Channel A Shutdown Pin (active-high, pulled-down)

A single voltage rail input ranging from 20V to 30V is provided to the power module. The power module can provide up to two output voltage rails to Pollux. For the sake of clarity, these channels are called A et B. For each of these channels, two signals control the activity of the supplies:

- **Enable signal**: Nominal start/stop signal that controls the state of the DC/DC converters.

- **Shutdown signal**: Emergency stop signal that electronically disconnect the load from the supply.

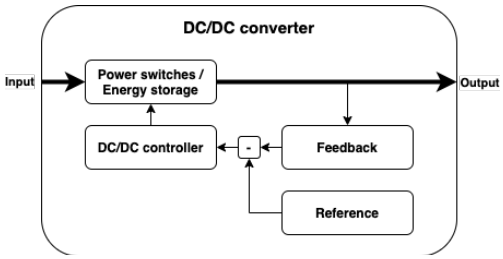
3.3.2. Architecture

One might find tempting to simply connect a DC/DC converter between the input and output pins of the power interface. However, an additional complexity arises once the requirements for scalability and redundancy are considered.

Indeed, when thinking system-wise, multiple Pollux III motherboards will be stacked next to each other, effectively connecting the outputs of multiple power modules together.

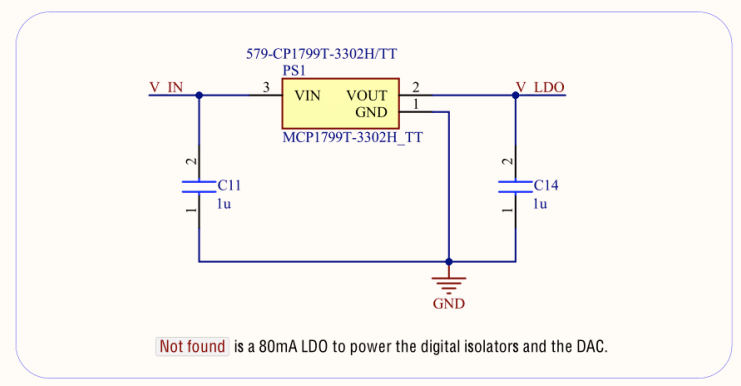
With a naïve architecture, if only one supply fails in short-circuit, all boards are automatically also short-circuited, which at best result in a very high current dissipation and heat generation in the boards and at worst, creates a chain reaction which destroys the power modules one by one.

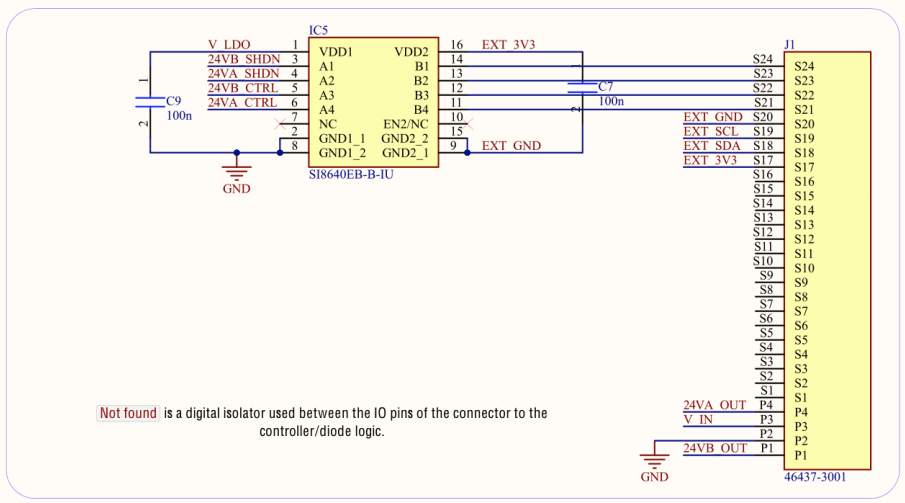
To understand the situation,

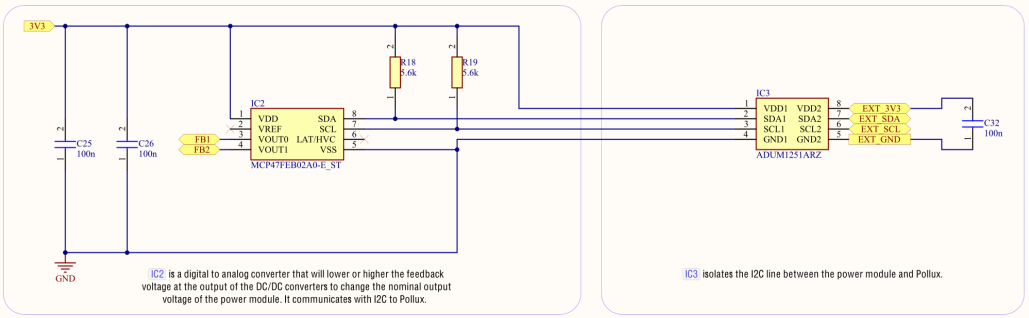


3.3.3. Control signals

To power the internal systems in the power module, a 3.3V CP1799T LDO is used. It is sourced from the input voltage, which can range from 21V to 29V and provides a stable 3.3V output to the digital isolators and the DAC.



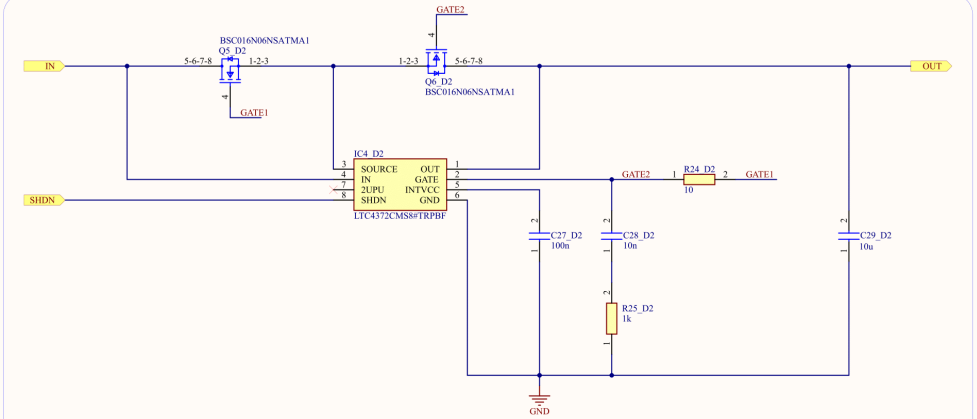




IC2 is a digital to analog converter that will lower or higher the feedback voltage at the output of the DC/DC converters to change the nominal output voltage of the power module. It communicates with I2C to Pollux.

IC3 isolates the I2C line between the power module and Pollux.

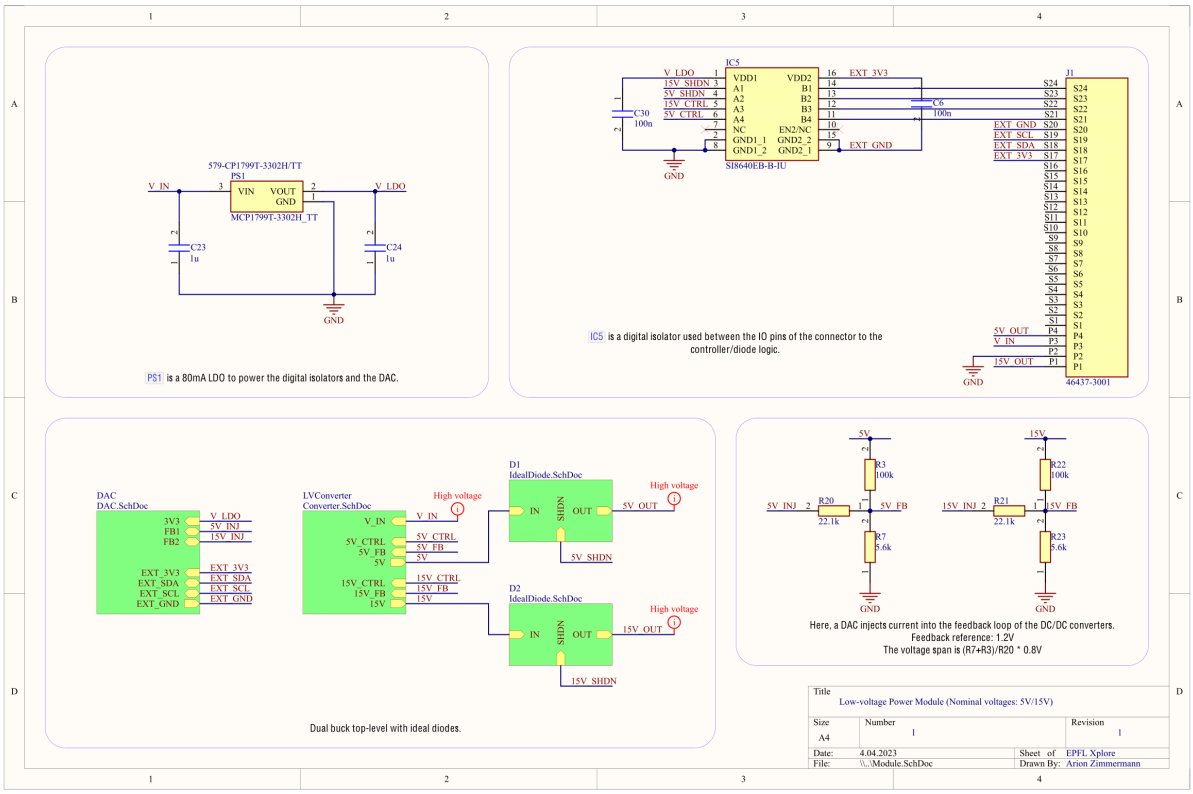
Title	Isolated Digital-to-Analog Converter		
Size	Number		Revision
A4	1		1
Date:	4.04.2023	Sheet of	EPFL Xplore
File:	\\..\DAC.SchDoc	Drawn By:	Arion Zimmermann

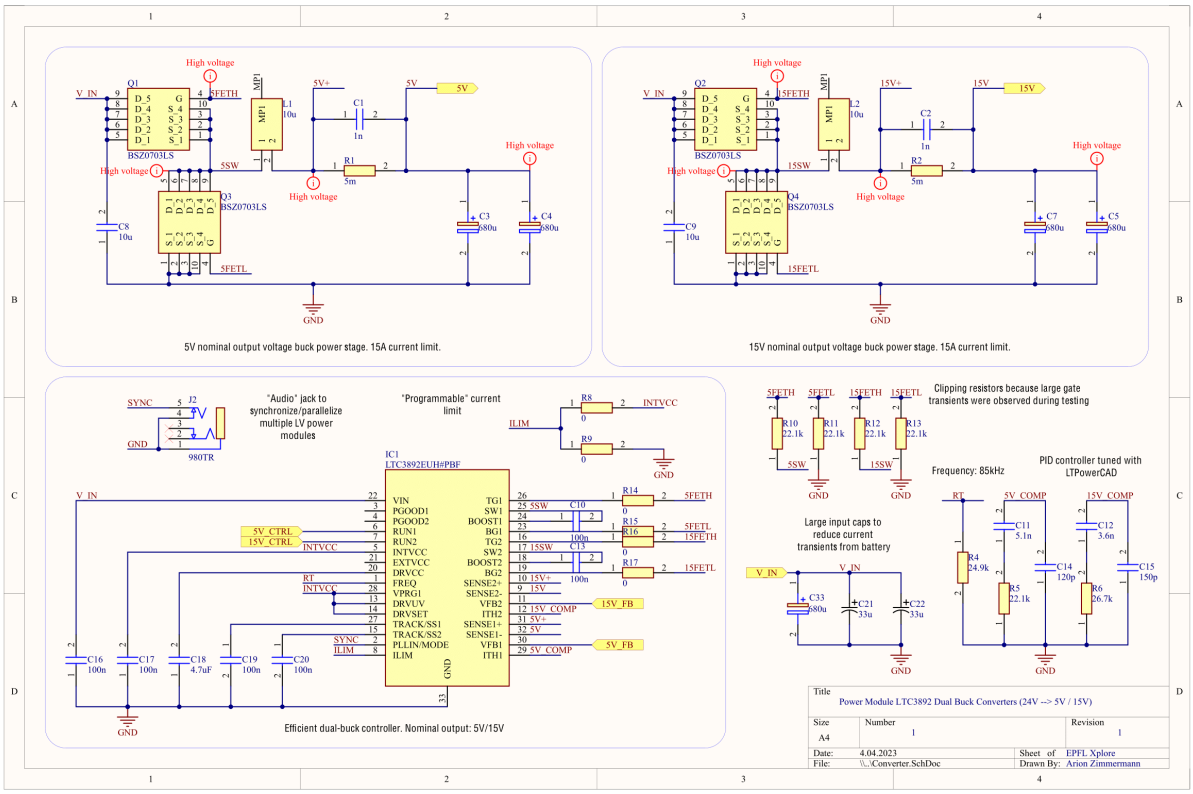


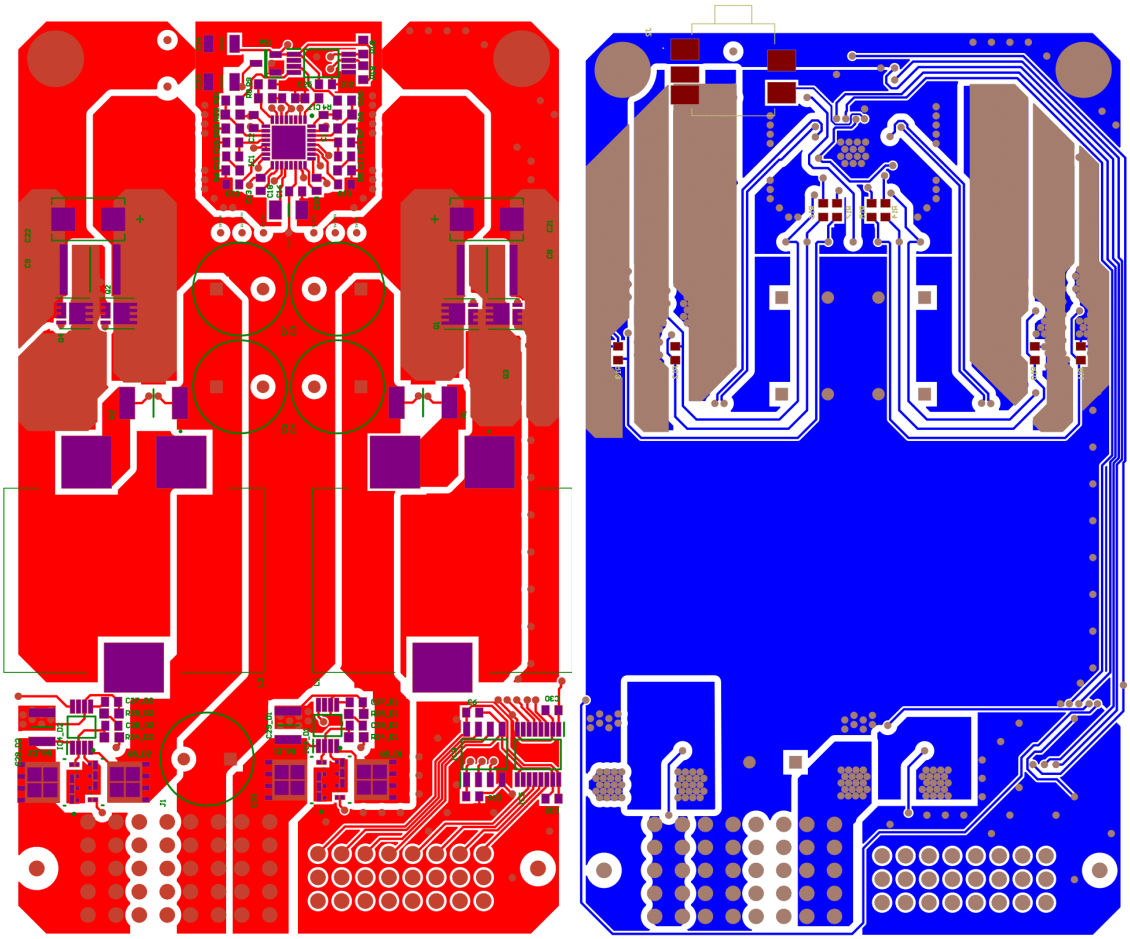
The two MOSFETS Q5 and Q6 control the current from the DC/DC converter and the load. Having an output voltage higher than the input voltage will trigger the IC4 ideal diode controller to disconnect the DC/DC converter from the load line. Some care was taken to reduce transients on the gate pin.

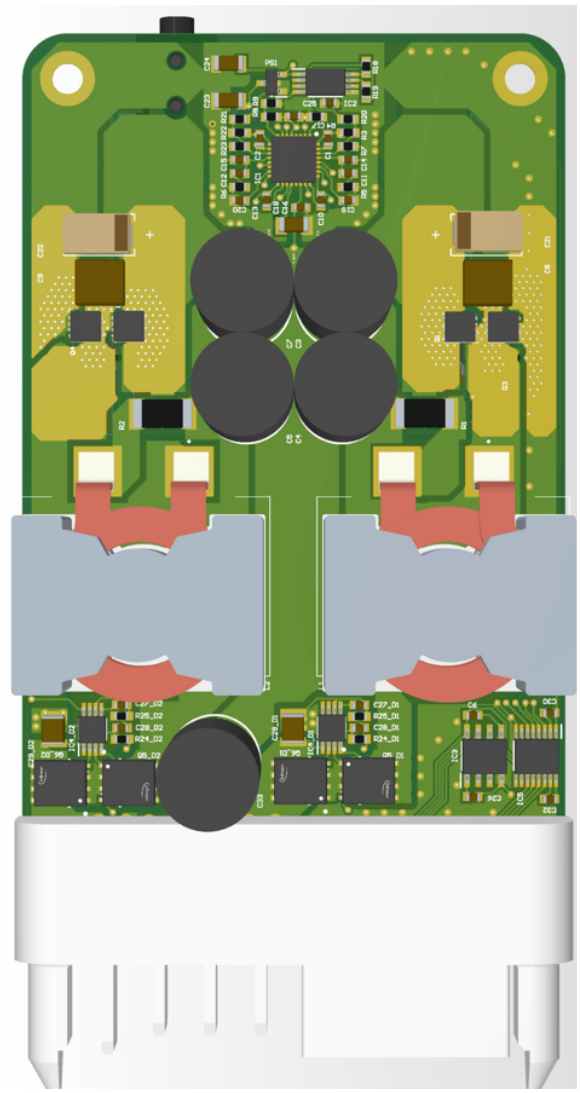
Title	Ideal Diode		
Size	Number		Revision
A4	1		1
Date:	4.04.2023	Sheet of	EPFL Xplore
File:	\\..\IdealDiode.SchDoc	Drawn By:	Arion Zimmermann

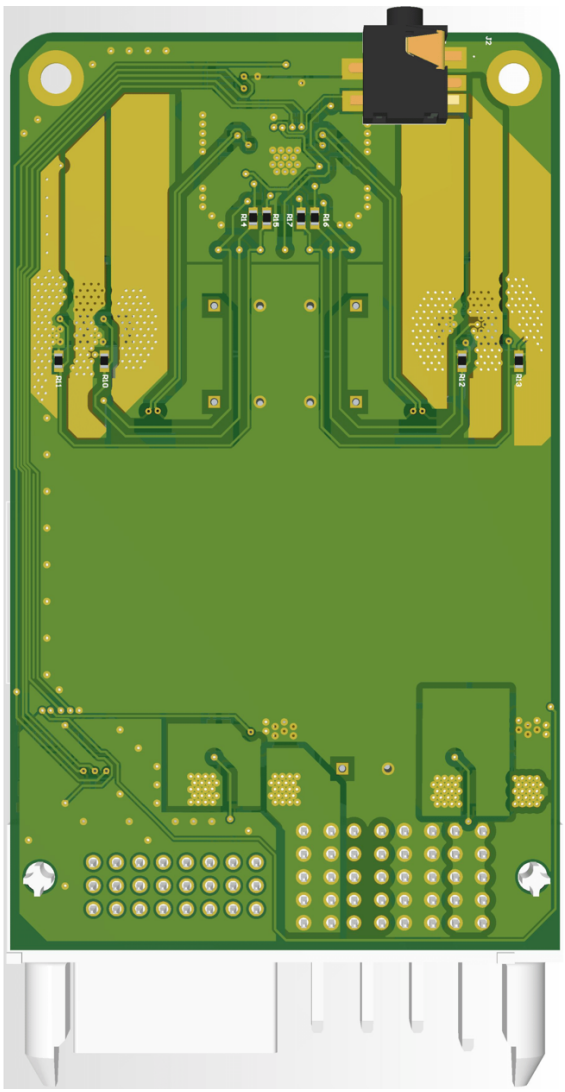
3.4. Low-voltage power module



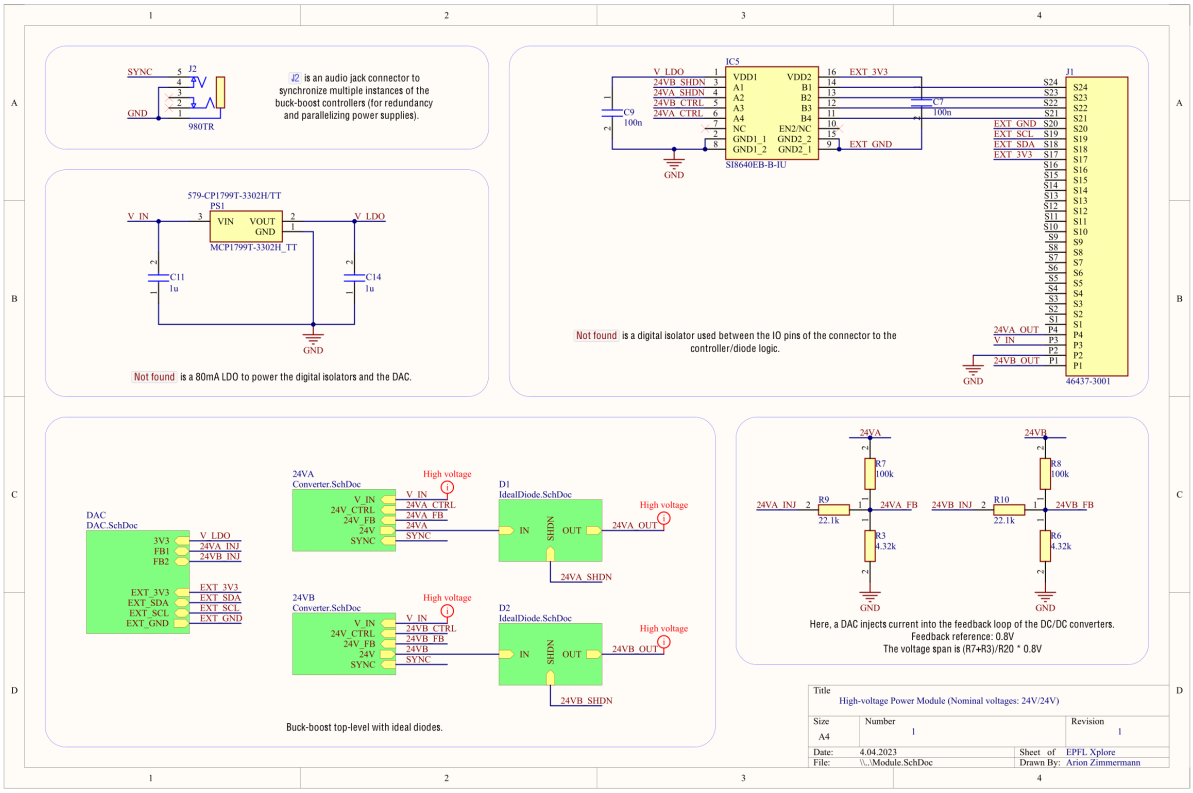


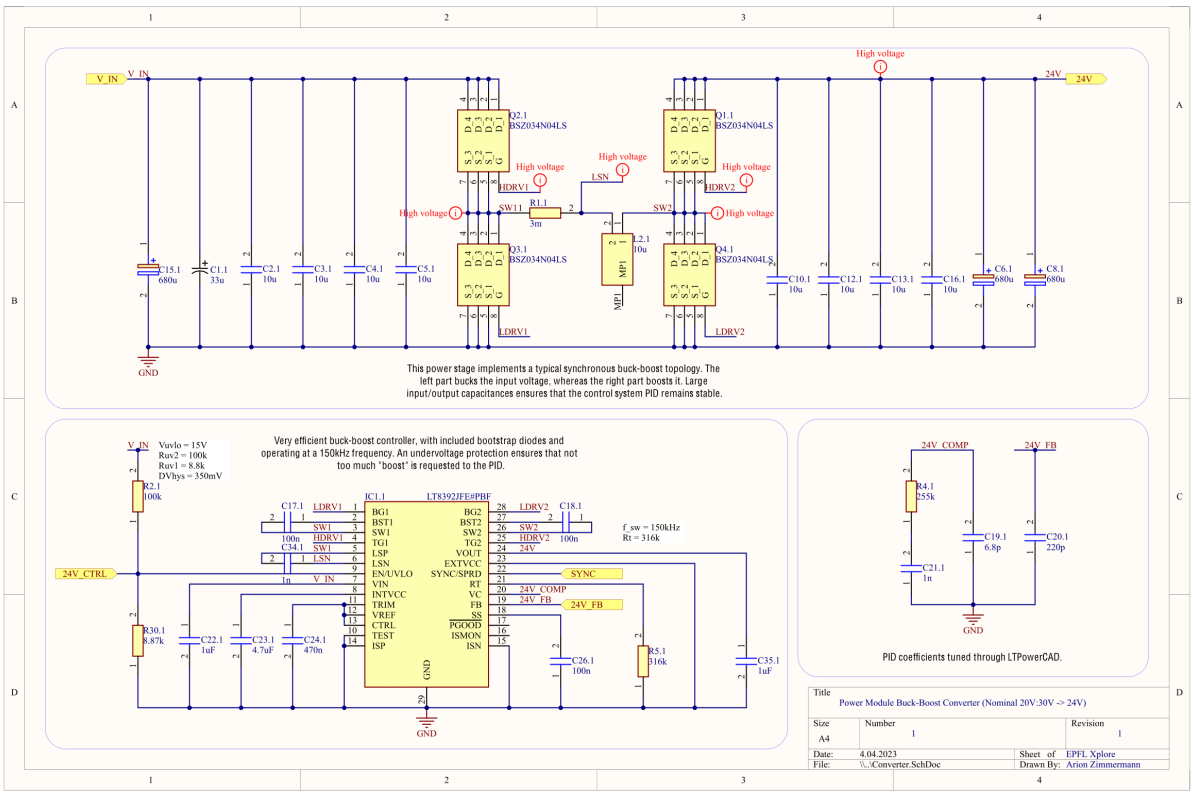


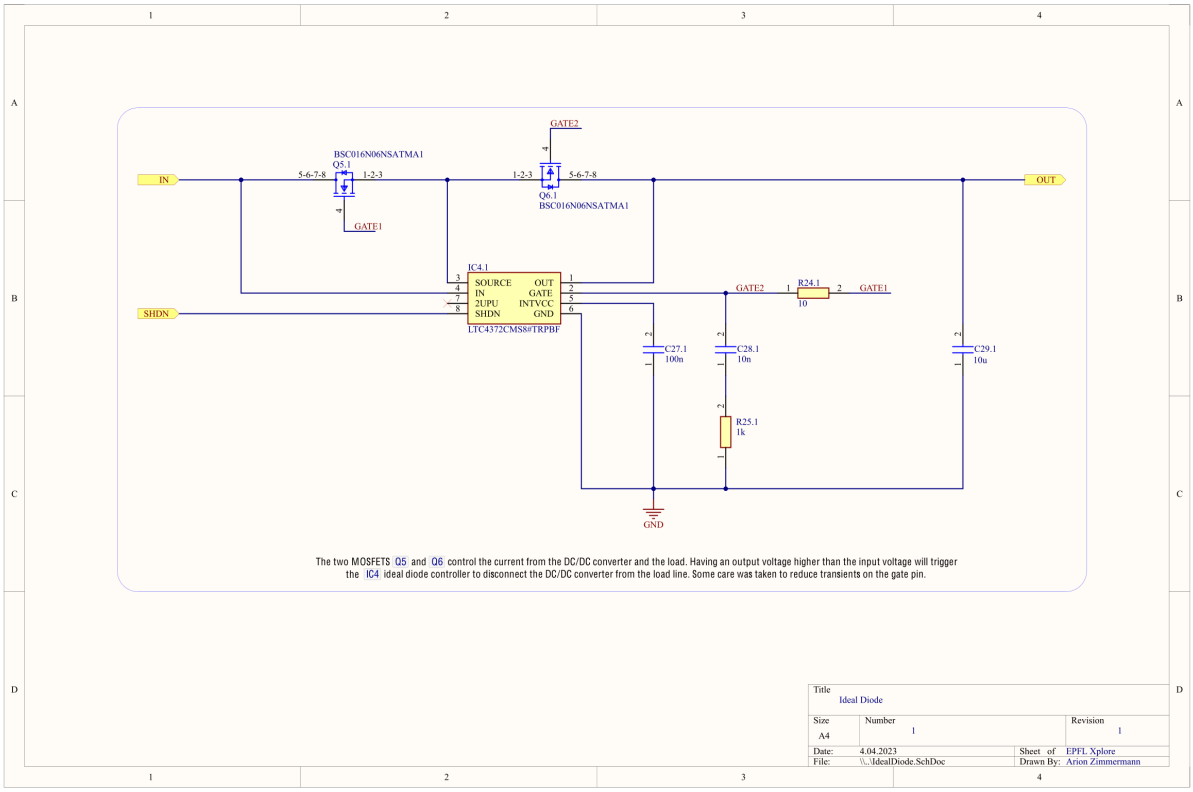




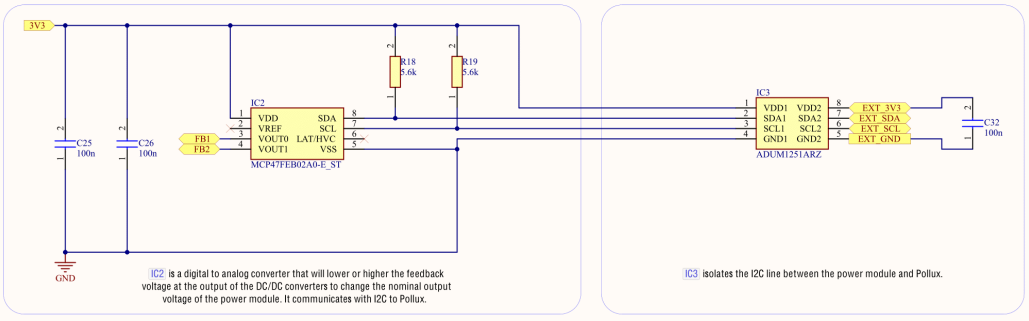
3.5. High-voltage power module



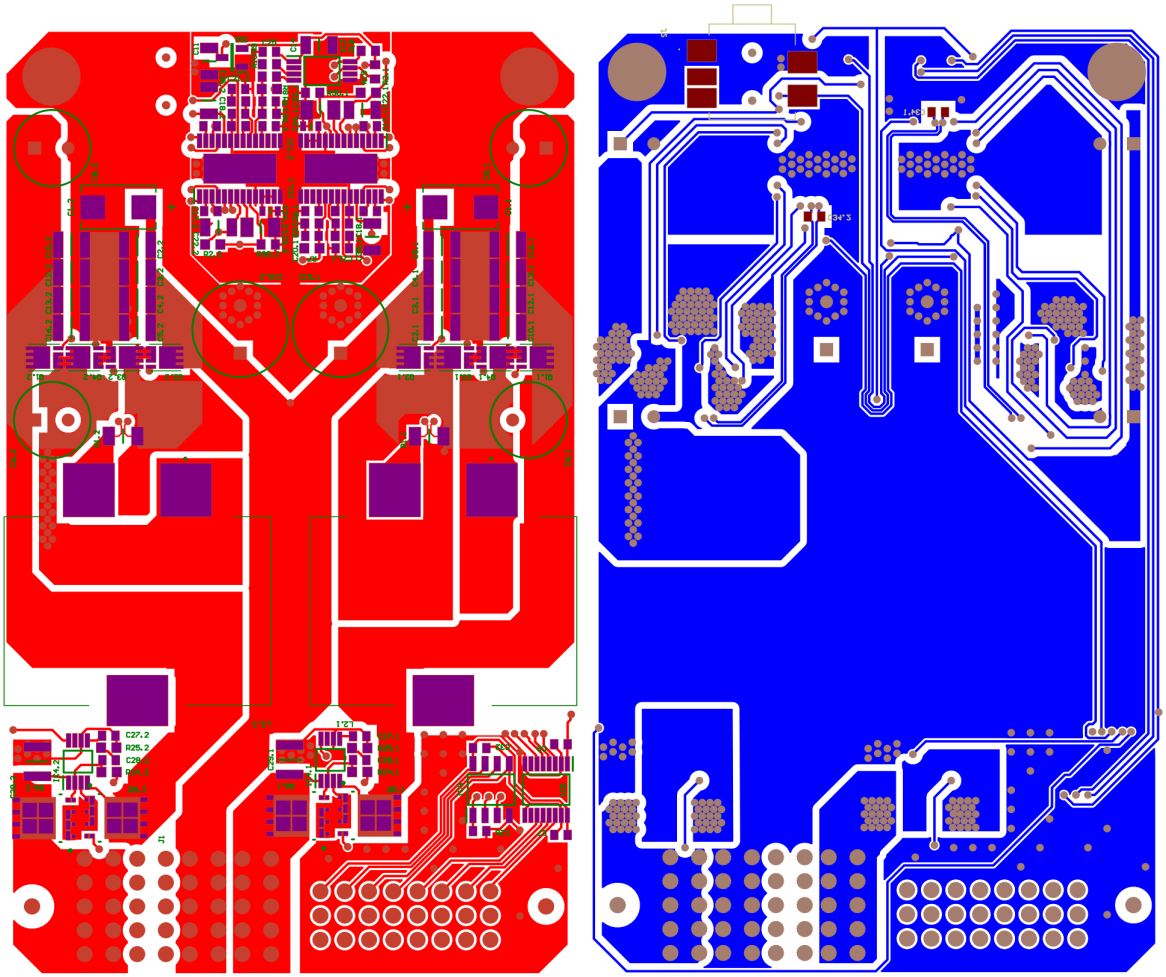


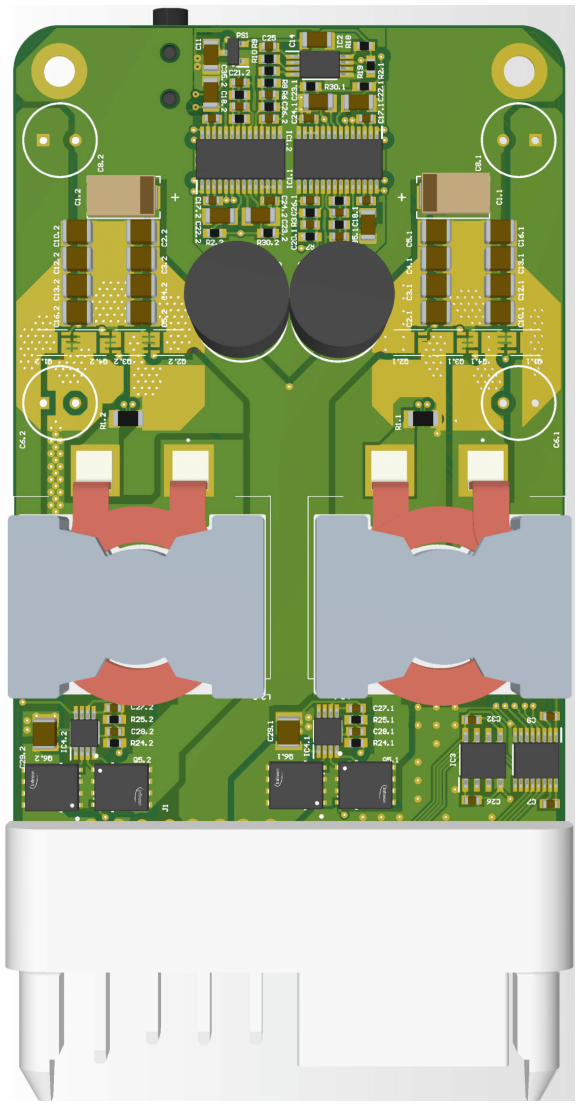


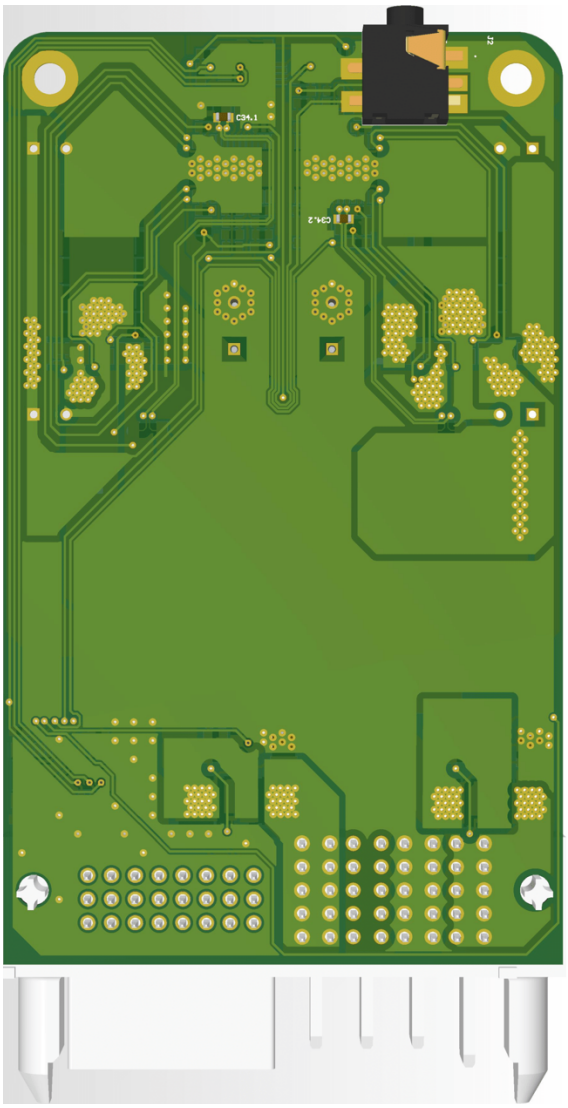
Title	Ideal Diode	
Size: A4	Number: 1	Revision: 1
Date: 4.04.2023	Sheet of EPFL Xplore	
File: \\...\IdealDiode.SchDoc	Drawn By: Arion Zimmermann	



Title	Isolated Digital-to-Analog Converter	
Size: A4	Number: 1	Revision: 1
Date: 4.04.2023	Sheet of EPFL Xplore	
File: \\...\DAC.SchDoc	Drawn By: Arion Zimmermann	



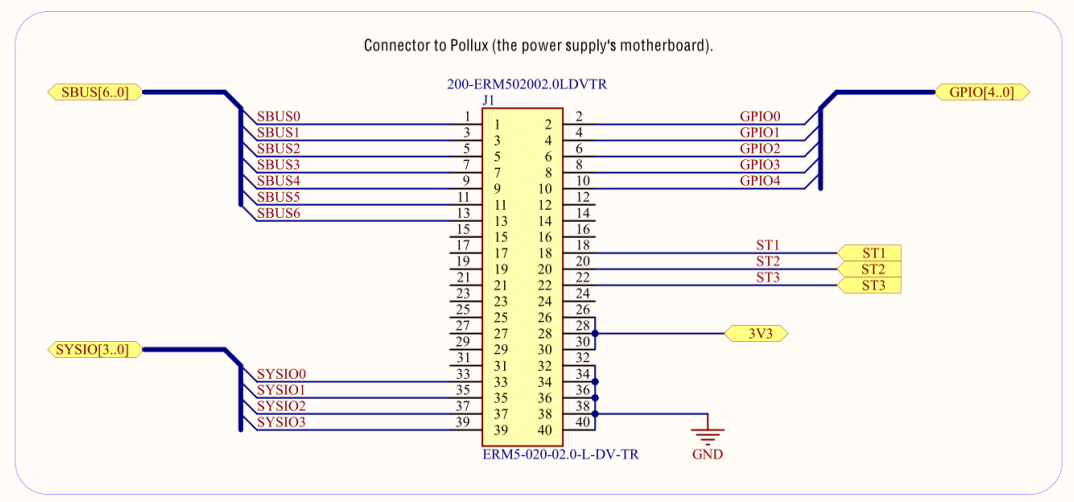




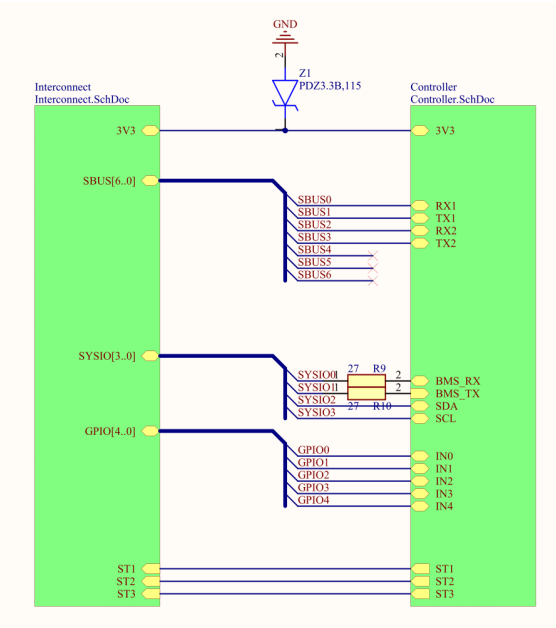
3.6. Castor II

A physical separation exists between the power system's motherboard and the wireless control module. Those two boards are interconnected through a 40-pins connector that allows signals to be transmitted from one board to another. The pin assignment for this connector is very generic to allow forward-compatibility and is described in Appendix A. This bus consists of several subinterfaces: a sensor bus, a system bus, a general purpose bus and a status bus.

Schematic @ corresponds to this interface and is shown below for a better understanding.



Eventually, the interface must be connected to a real hardware implementation of the WiFi module. Schematics @ gives hardware functions to the interface pins. Some additional safety is added before interfacing with Pollux, such as a Zener diode, which clamps voltage transients to a 3.3V maximum voltage. Two 27Ω resistors are added to the BMS UART connection to prevent damage in case of a reverted connection with the BMS.



3.6.1. Sensor bus interface definition

The link between the wireless control board and the motherboard is implemented through the UART protocol. Up to two UART ports can be used for transmission. This dual port feature was used with Pollux II but is deprecated in Pollux III, where only the UART port 1 is significant.

Pin	**Corresponding function**
Sensor bus signal 0 (SBUS0)	Receive on UART RX port 1
Sensor bus signal 1 (SBUS1)	Transmit on UART TX port 1
Sensor bus signal 2 (SBUS2)	Receive on UART RX port 2
Sensor bus signal 3 (SBUS3)	Transmit on UART TX port 2
Sensor bus signal 4 (SBUS4)	*Unused (legacy)*
Sensor bus signal 5 (SBUS5)	*Unused (legacy)*
Sensor bus signal 6 (SBUS6)	*Unused (legacy)*

3.6.2. System bus interface

This interface is used to connect the power supply to the rest of the Rover. In particular, it consists of an UART link to connect to the Rover's BMS (battery management system) and an I^2C connection to the power system's sensors. Once again, the pin assignments are stated below.

Pin	**Corresponding function**
System bus signal 0 (SYSIO0)	UART RX
System bus signal 1 (SYSIO1)	UART TX
System bus signal 2 (SYSIO2)	I^2C SDA
System bus signal 3 (SYSIO3)	I^2C SCL

3.6.3. General purpose bus interface

Five general purpose lines also connect the control module to the power module. Those are mainly used for debugging purposes. No special protocol is defined for those general purpose signals.

3.6.4. Status bus interface

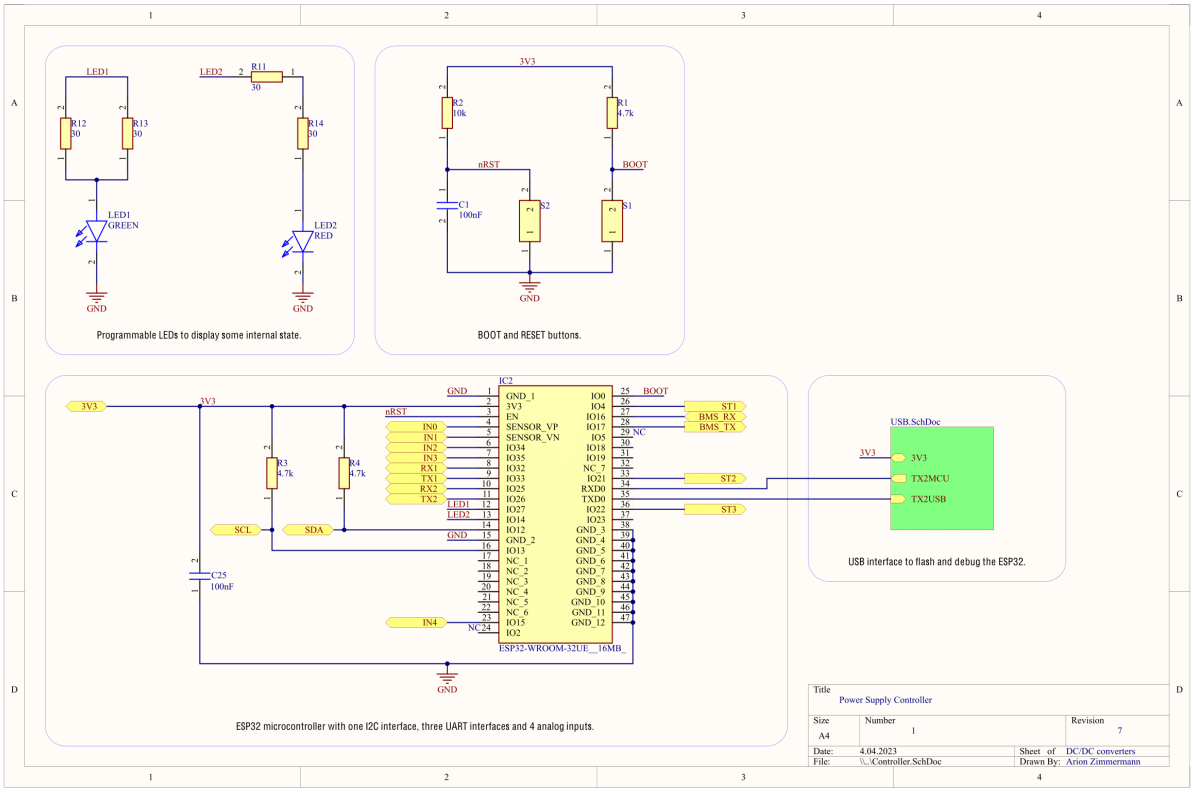
To indicate the state of the control module, three signals connect the control module to the power module. These lines shall carry PWM signals whose duty cycle completely defines the state of the control module.

3.6.5. Power interface

A 3.3V bi-directional power port connects Castor to Pollux. One the one hand, if Castor is powered by USB, Pollux also receives a 3.3V input supply. On the other hand, if Pollux is powered by the battery, Castor receives a 3.3V input. The current specification for the power port is 2A maximum.

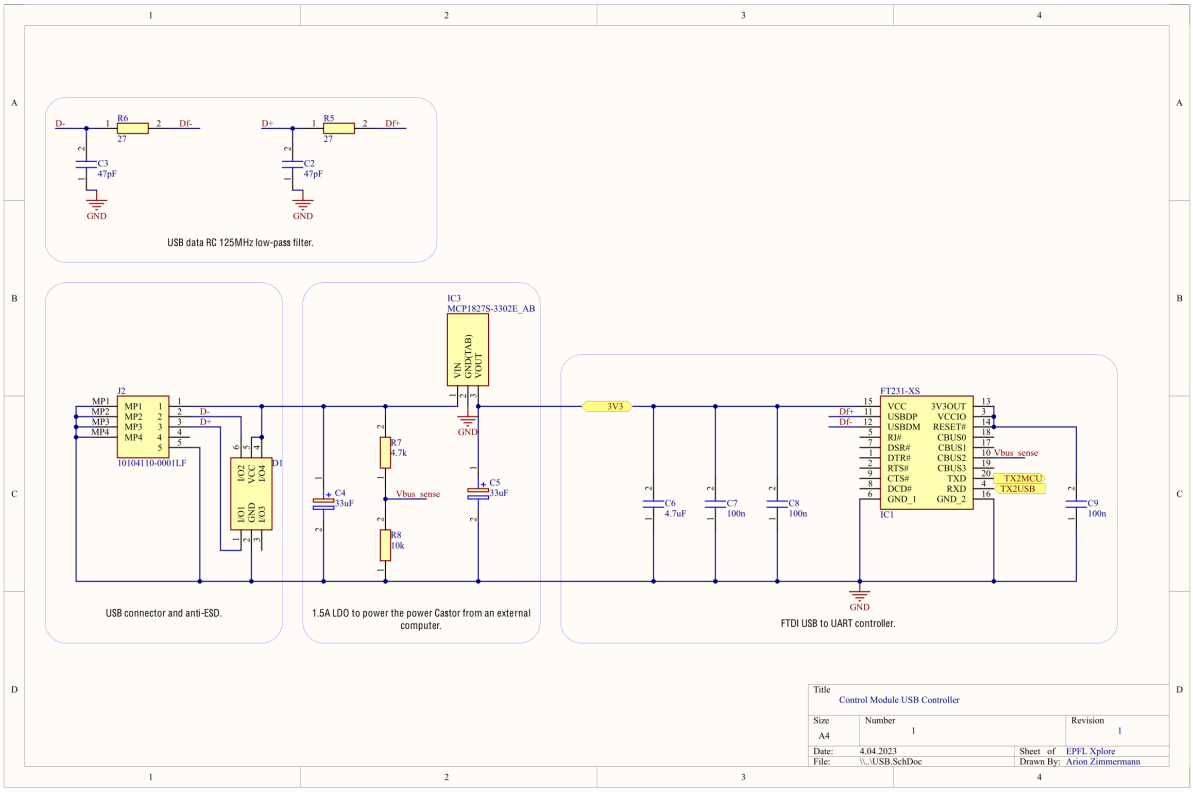
3.6.6. Wireless module

An ESP32-WROOM-32UE 2.4 GHz WiFi-enabled module from Espressif Systems is used to interface with Pollux and allow Castor to communicate with the end-user.
A 100nF bypass capacitor is placed on the power input line, as recommended by the datasheet and a reset button with a 1kHz low-pass filter allows the end-user to send an active-low reset signal to the ESP32.
Additionally, the ESP32's bootloader settings can be changed by using the active-low boot button.
Two programmable LEDs (green and red) are also present to indicate Castor's state. Note that the lines IN0, IN1, IN2 and IN3 are input only lines, whereas IN4 is an I/O line.
Schematic @ represents the conceptual connections between the ESP32, the LEDs and the buttons.



3.6.7. Programming interface

To program the ESP32, an FT231XS USB-to-UART bridge is embedded in Castor. This IC translates the USB2 differential signals D+ and D- into the UART RX and TX signals. Schematics @ shows how this part of the system was implemented. Note that, in general, a FT231 driver must be installed on the target operating system to communicate with Castor.



A standard Micro USB type B connector is used to connect the Castor board with a host computer.

Several TVS diodes are connected in parallel to the connector's signal lines, to prevent unintentional electrostatic discharges when touching the connector or plugging it into the host computer. These TVS diodes can be considered as back-to-back high-energy Zener diodes that will conduct once the voltage on any line exceeds 3.3V, preventing damage to sensitive components on the board.

In addition to this, 125MHz RC low-pass filters are added on the signal lines D+ and D-, so reject high-frequency noise as much as possible before feeding the signals into the FT231XS IC. This low-pass filter is recommended in the FT231-XS datasheet.

The host computer's 5V/1A USB power output is used to power-up the ESP32 and the USB-to-UART bridge without the need of Pollux' power. Nevertheless, the 5V power line must be converted to a stable 3.3V line. This feature is implemented by the MCP1827 LDO. When connected to Pollux, this LDO can also provide current to the whole power system.

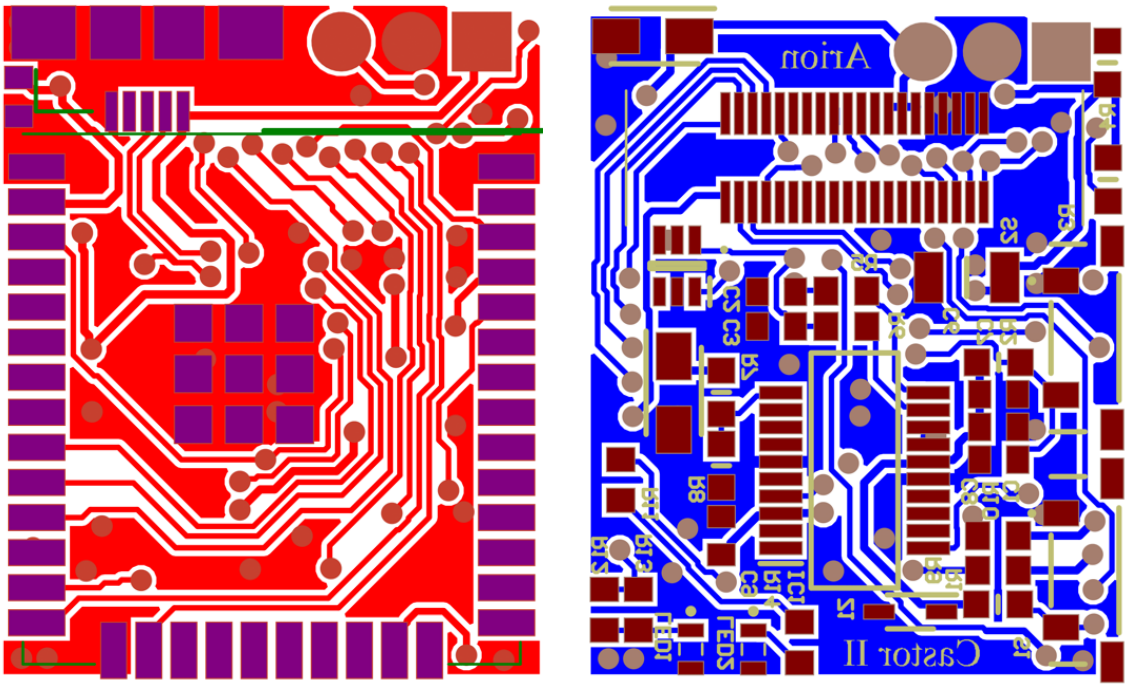
33µF Tantalum-Polymer capacitors are added on the input and output side of the LDO to stabilize it and compensate possible voltage transients on the lines. Tantalum-Polymer capacitors are used here for they have a high capacity to volume ratio, which is critical for the compactness of Castor. In general, Tantalum-Polymer capacitors exhibit a low equivalent series resistance (ESR), which is a good marker to optimize the efficiency of a

switching system. In the case of LDOs however, it might negatively avert the stability of the voltage regulator. Care must be taken to ensure that the output capacitance's ESR is higher than the one required in the MCP1827 datasheet. @

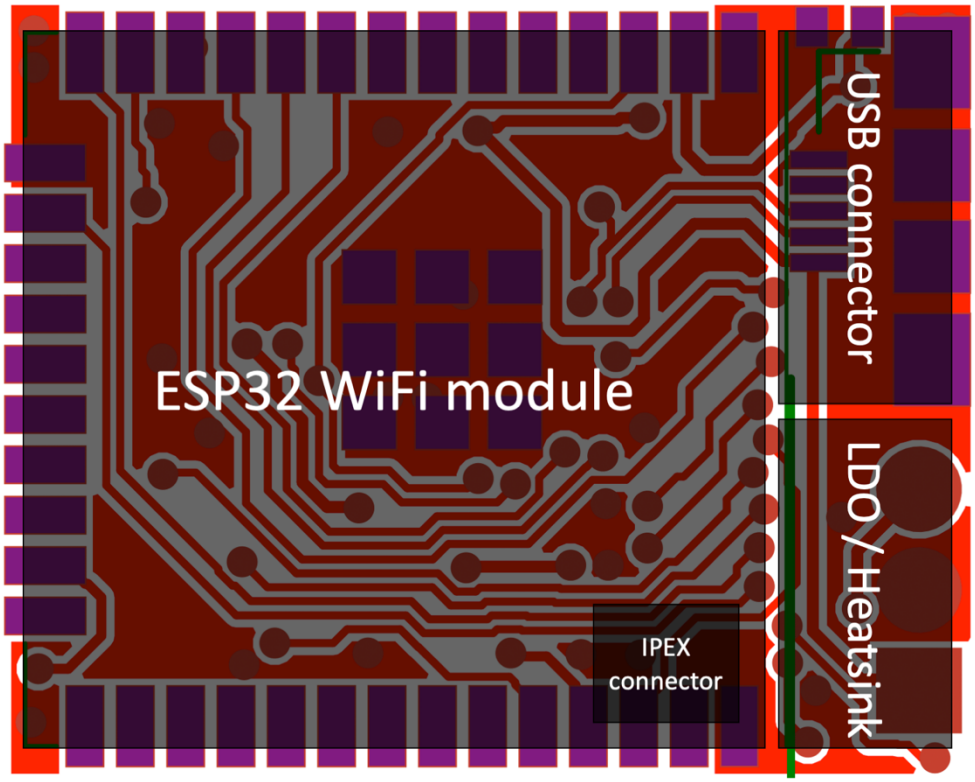
The FT231XS IC is used in a self-powered configuration (see section 6.2 on its datasheet). Bypass capacitors are added around the FT231 chip, as recommended by the datasheet.

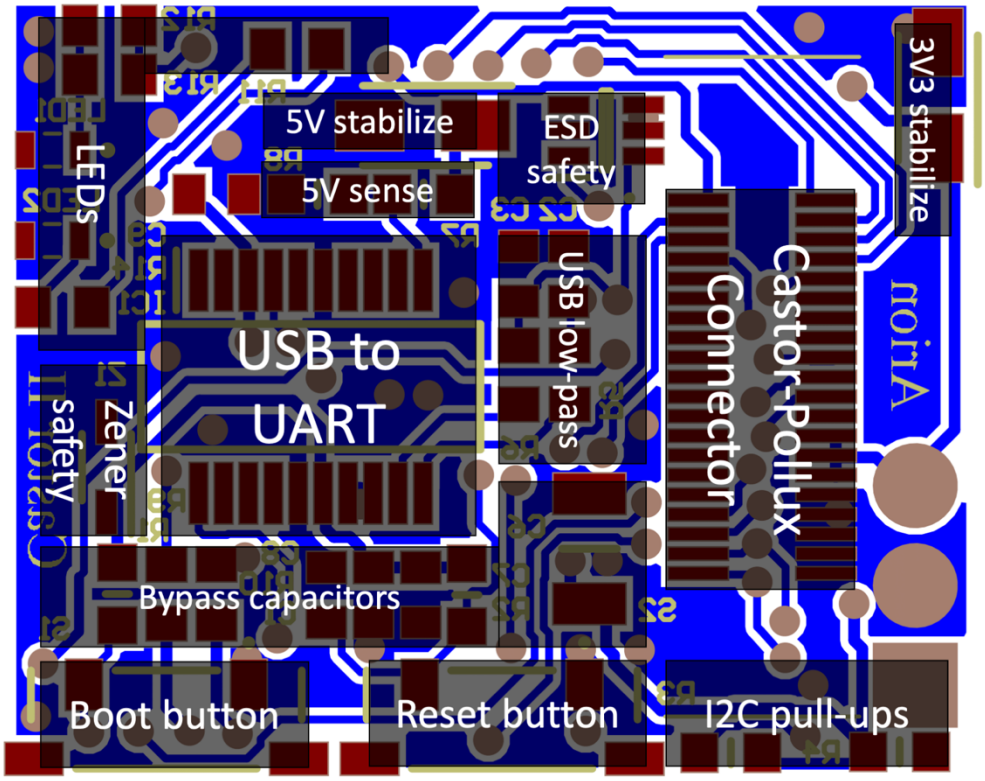
3.6.8. Layout

Layout @ shows how the schematics were physically implemented into a very compact, 2-layers and easy to manufacture PCB.



All components with may require an interface with the rest of the system were placed on the upper side of the board, whereas the sensitive ICs which are used to control the ESP32 are placed on the bottom side, as shown in diagrams @ and @ respectively.



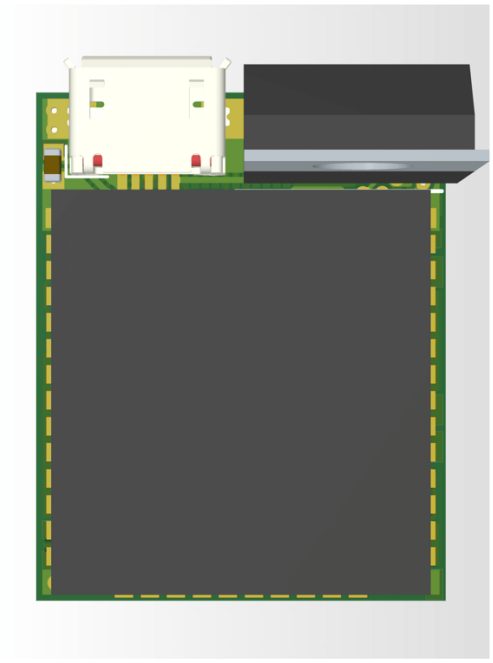


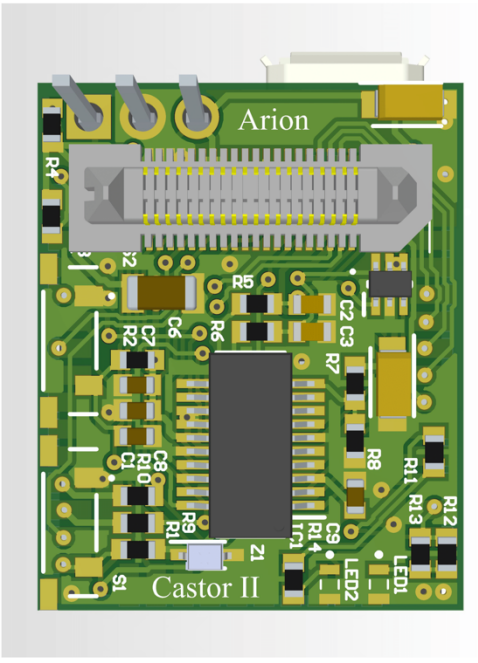
The PCB layout was routed to follow the design rules prescribed in the Appendix. In addition to the design rules, care was taken to route together the D+/D- and Df+/Df- differential lines as tightly as possible. Nevertheless, it is not crucial to route these

differential pairs extremely well, since the transmission speed through these lines is mainly limited by the conversion to UART made by the FT231 IC.

Note that an external IPEX antenna is needed for the ESP32 to function properly. Otherwise, permanent damage could be made to the Castor board when transmitting to a radio power level higher than 13dBm.

Render @ is the 3D view of the Castor II PCB.





4. Software

4.1. Pollux III software

Pollux III runs on an STM32H750 microcontroller. All related software was developed using STM32 CubeIDE 1.13.0 and its corresponding firmware/BSP. FreeRTOS is used as an operating system. The CMSIS V1 hardware abstraction interface is used to bind FreeRTOS to the ARM Cortex processor architecture. Compared to bare-bones programming, using an operating system such as FreeRTOS with CMSIS V1 is useful to allow a pseudo-parallel execution of multiple tasks. Section 0 describes the *System Thread* abstraction layer developed above the FreeRTOS operating system to simplify the development, maintenance, and testing of software for Pollux III.
The hierarchy of the Pollux III CubeIDE project is decomposed as follows:

1. **System**: Contains the vast majority of Pollux III software. The folder itself only contains the *System Thread* abstraction layer. Its subfolders form a secondary hierarchy:

    a. **Debug**: Puts at disposal debugging and profiling tools that were developed over my EPFL years.

    b. **GUI**: Contains the Graphical User Interface that allows the microcontroller to display states, events, and measurements data, as well as to receive user input from the touch screen to restart some power supplies.

    c. **Libraries**: Englobes all the Pollux III software that could be fully isolated from the "main" code. Each library fulfils a specific and independent function.

    d. **Logging**: Allows Pollux III to store mission data on a persistent storage.

    e. **Misc**: Stores the miscellaneous code that does not play an important role.

    f. **Sensors**: Manages the Pollux III sensors.

    g. **Supplies**: Manages the Pollux III supplies.

2. **Core**: Holds the main code and peripheral initialization code, mainly generated by CubeIDE and bootstraps the launch of the different *System Threads* in **System**.

3. **Drivers**: Encompasses the BSP for the STM32H750 microcontroller, as well as the CMSIS V1 ARM hardware interface.

4. **Middlewares**: Contains the FreeRTOS sources.

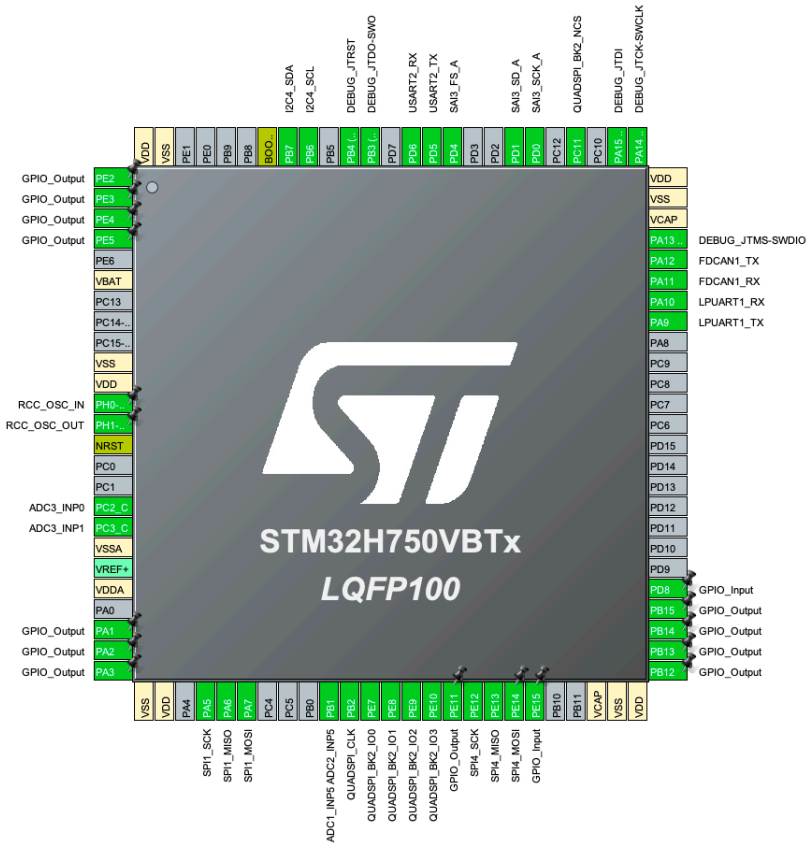
5. **Pollux III.ioc**: Represents the physical hardware configuration of a microcontroller. It is used to configure the pins to their correct function, as well as configuring the system clock. The system hardware configuration is described in section 4.1.1.

6. **Pollux III.launch**: Describes the debugging configuration that is used when flashing the Pollux III software on the STM32H750 microcontroller through a JTAG/SWD connection. It is usually edited through Run > Run configurations… or Run > Debug configurations… in CubeIDE.

7. **STM32H750VBTX_FLASH.ld**: Automatically generated linker script for GCC. Represents how the data and code are fit into the microcontroller's memory regions. Changes to this file may be useful in last resort to access specific memory regions that cannot be accessed otherwise.

4.1.1. Hardware configuration

CubeIDE provides a software configuration tool (previously known Cube MX). The microcontroller pins can be configured from a graphical user interface to the corresponding function, as defined when designing the Pollux III PCB. Figure @ depicts the used pin configuration and table @ summarizes the activated function for the pins used.

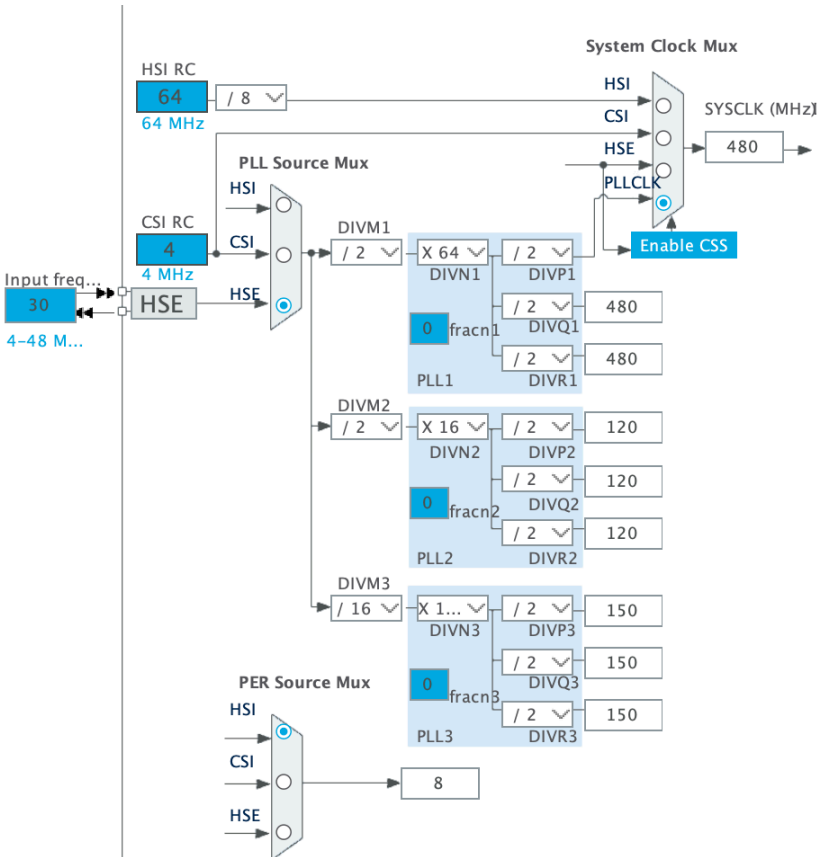


Pin name	Unit	Description
PE2	GPIO	HVB shutdown pin
PE3	GPIO	HVA shutdown pin
PE4	GPIO	HVB enable pin
PE5	GPIO	HVA enable pin
PH0	OSC	High-speed oscillator (30MHz) input
PH1	OSC	High-speed oscillator (30MHz) output

PC2	ADC3	Measures a scaled version of the battery voltage
PC3	ADC3	Measures the input 3V3 voltage
PA1	GPIO	Address bit 0 for the power monitor multiplexer
PA2	GPIO	Address bit 1 for the power monitor multiplexer
PA3	GPIO	Address bit 2 for the power monitor multiplexer
PA5	SPI1	SCK SPI clock to the power monitors
PA6	SPI1	MISO SPI master input from the power monitors
PA7	SPI1	MOSI SPI master output to the power monitors
PB1	ADC1	*Unused (Legacy)*
PB2	QSPI2	CLK QuadSPI clock to the external flash memory
PE7	QSPI2	QuadSPI lane 0
PE8	QSPI2	QuadSPI lane 1
PE9	QSPI2	QuadSPI lane 2
PE10	QSPI2	QuadSPI lane 3
PE11	GPIO	Touchscreen active-low chip select
PE12	SPI4	SCK SPI clock to the touchscreen
PE13	SPI4	MISO master input from the touchscreen
PE14	SPI4	MOSI master output to the touchscreen
PE15	GPIO	Interrupt lane from touchscreen
PB12	GPIO	LVB shutdown pin
PB13	GPIO	LVA shutdown pin
PB14	GPIO	LVB control pin
PB15	GPIO	LVA control pin
PD8	GPIO	*Unused (Legacy)*
PA9	LPUART1	TX (TX to computer) for the JTAG debugger
PA10	LPUART1	RX (TX to microcontroller) for the JTAG debugger
PA11	FDCAN	RX for flexible-datarate CAN bus controller
PA12	FDCAN	TX for flexible-datarate CAN bus controller
PA13	JTAG/SWD	JTMS (state-machine control) for the JTAG debugger
PA14	JTAG/SWD	JTCK (clock) for the JTAG debugger
PA15	JTAG/SWD	JTDI (data input) for the JTAG debugger
PC11	QSPI2	nCS active-low chip select for the QuadSPI unit
PD0	SAI3	*Unused (Legacy)*
PD1	SAI3	*Unused (Legacy)*
PD4	SAI3	*Unused (Legacy)*
PD5	USART3	TX (TX to WiFi module) for the WiFi module
PD6	USART3	RX (TX to microcontroller) for the WiFi module
PB3	JTAG/SWD	JTDO (data output) for the JTAG debugger
PB4	JTAG/SWD	JTRST (reset) for the JTAG debugger

PB6	I2C4	SCL line for the I2C communication to the power supplies
PB7	I2C4	SDA line for the I2C communication to the power supplies

In addition to pin configuration, it is necessary to configure the different clock speeds for each unit. Figure @ illustrates which clock speeds were selected for the main Phase Locked Loops (PLLs). All units have a base clock speed derived from these PLLs.



Here, the High-Speed Internal oscillator runs at 64MHz but is not used because its stability is not high enough to achieve optimal performance on the STM32H750 microcontroller. Nevertheless, a High-Speed External oscillator, set at 30MHz, is used as a base clock. Nine PLLs are defined at 480MHz, 120MHz and 150MHz depending on which hardware unit requires them.

4.1.2. Thread management

The *System Thread* abstraction layer makes use of C++ classes to simplify the FreeRTOS thread management by encapsulating the Thread lifecycle management in an Arduino-like code structure. Before detailing this abstraction layer, it is necessary to understand the fundamentals of a real-time operating system, such as FreeRTOS.

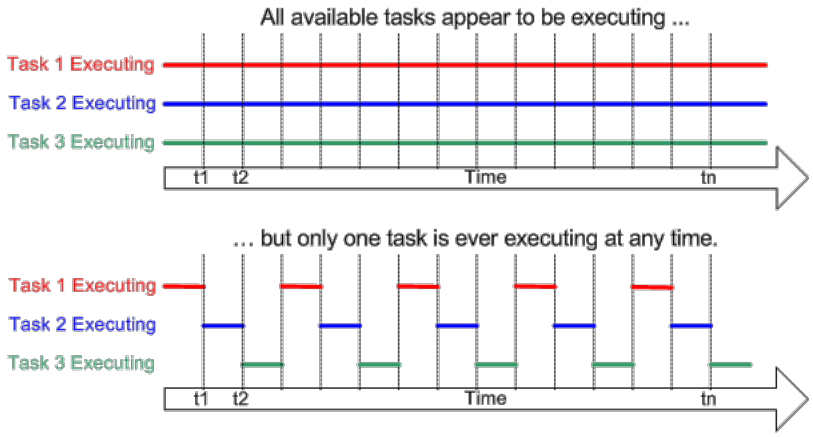
4.1.2.1. Foundations of FreeRTOS

This section is mainly based on the FreeRTOS documentation and directly uses figures and explanations copy-pasted from it.

The **kernel** is the core component within an operating system. Operating systems such as Linux employ kernels that allow users access to the computer seemingly simultaneously. Multiple users can execute multiple programs apparently concurrently.

Each executing program is a **task** (or thread) under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be **multitasking**.

A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it **appear** as if each task is executing concurrently. This is depicted by the diagram below which shows the execution pattern of three tasks with respect to time. The task names are colour coded and written down the left hand. Time moves from left to right, with the coloured lines showing which task is executing at any time. The upper diagram demonstrates the perceived concurrent execution pattern, and the lower the actual multitasking execution pattern.



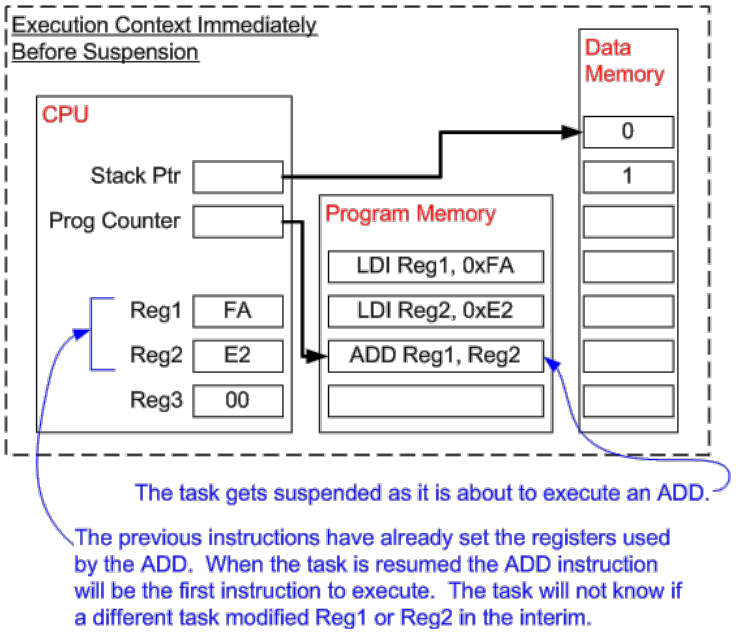
The **scheduler** is the part of the kernel responsible for deciding which task should be executing at any particular time. The kernel can suspend and later resume a task many times during the task lifetime.

The **scheduling policy** is the algorithm used by the scheduler to decide which task to execute at any point in time. The policy of a (non real time) multi user system will most

likely allow each task a "fair" proportion of processor time. The policy used in real time / embedded systems is described later.

In addition to being suspended involuntarily by the kernel a task can choose to suspend itself. It will do this if it either wants to delay (**sleep**) for a fixed period, or wait (**block**) for a resource to become available (e.g. a serial port) or an event to occur (eg a key press). A blocked or sleeping task is not able to execute, and will not be allocated any processing time.

As a task executes it utilizes the processor / microcontroller registers and accesses RAM and ROM just as any other program. These resources together (the processor registers, stack, etc.) comprise the task execution **context**.



A task is a sequential piece of code - it does not know when it is going to get suspended (swapped out or switched out) or resumed (swapped in or switched in) by the kernel and does not even know when this has happened. Consider the example of a task being suspended immediately before executing an instruction that sums the values contained within two processor registers. While the task is suspended other tasks will execute and may modify the processor register values. Upon resumption the task will not know that the processor registers have been altered - if it used the modified values the summation would result in an incorrect value.

To prevent this type of error it is essential that upon resumption a task has a context identical to that immediately prior to its suspension. The operating system kernel is responsible for ensuring this is the case - and does so by saving the context of a task as it is suspended. When the task is resumed its saved context is restored by the operating system kernel prior to its execution. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called **context switching**.

Real time operating systems (**RTOSes**) achieve multitasking using these same principles - but their objectives are very different to those of non real time systems. The different objective is reflected in the scheduling policy. Real time / embedded systems are designed to provide a timely response to real world events. Events occurring in the real world can have deadlines before which the real time / embedded system must respond and the RTOS scheduling policy must ensure these deadlines are met.

To achieve this objective the software engineer must first assign a priority to each task. The scheduling policy of the RTOS is then to simply ensure that the highest priority task that is able to execute is the task given processing time. This may require sharing processing time "fairly" between tasks of equal priority if they are ready to run simultaneously.

4.1.2.2. *System Thread* abstraction layer

`void systemd_init()` from *System.cpp* is called immediately after the BSP drivers have been initialized but just before the FreeRTOS kernel is started. The purpose of this function is threefold. Firstly, it is the *root* FreeRTOS task, meaning that the "main" code in this function will be executed immediately after the initialization of the FreeRTOS kernel. Secondly, it serves as centre point between all *System Threads* that must be run simultaneously on the microcontroller, greatly simplifying the inter-thread communication. Thirdly, it is a function with a signature `extern void(void),` which allows it to be called from C code in *FreeRTOS.c*, but still be embedded in a C++ context.
A *System Thread* is a C++ class that inherits the Thread class defined in *Thread.h*. Once a *System Thread* is instantiated in `systemd_init()`, its parent constructor *Thread::Thread* is called and a FreeRTOS task is defined. When the scheduler launches the task, it calls the `virtual void init()` method once and then runs the `virtual void loop()` method in a loop.
Depending on the desired FreeRTOS task configuration, a few constructors can be used:

```
Thread(const char* name);
Thread(const char* name, osPriority priority);
Thread(const char* name, uint32_t stackSize);
Thread(const char* name, osPriority priority, uint32_t stackSize);
```

The Thread's `name` argument is required for logging and debugging purposes. It should clearly indicate the *System Thread's* purpose.
`priority` defines the FreeRTOS task priority for the *System Thread*. One of the following priorities defined by FreeRTOS can be used:

```
osPriorityIdle          = -3,          ///< priority: idle (lowest)
osPriorityLow           = -2,          ///< priority: low
osPriorityBelowNormal   = -1,          ///< priority: below normal
osPriorityNormal        =  0,          ///< priority: normal (default)
osPriorityAboveNormal   = +1,          ///< priority: above normal
osPriorityHigh          = +2,          ///< priority: high
osPriorityRealtime      = +3,          ///< priority: realtime (highest)
```

It is crucial to carefully select the thread priority, so that tasks that should react to events quickly or that must run at a precise loop rate have a higher priority.
`stackSize` defines the amount of stack reserved for a given *System Thread*. The stack stores all the local variables in the `void loop()` method and of all the subfunctions called by the loop. Managing the stack size is the cause of many troubles, and a good practice is to keep at the default value… If the Pollux III software starts crashing randomly during

development, it is a good idea to attempt changing the stack size of the *System Thread* in development. For debugging, a breakpoint can be inserted in the `void vApplicationStackOverflowHook()` and the `pxCurrentTCB` variable can be inspected [procedure].

Once a thread has finished running, `void terminate()` must be called to stop the *System Thread's* main loop. Usually, this method is not used since most *System Threads* are designed to run permanently.

Important note: The *System Thread* abstraction layer defines tasks that can be created and terminated dynamically through FreeRTOS. The *Idle* task, which has the lowest possible priority, is responsible for freeing the resources used by a terminated *System Thread*. It is therefore of high importance to allow the scheduler to run this Idle task by calling **`osDelay`** or entering a blocked state in each loop function in the Pollux III code.

`void setTickDelay(uint32_t ms)` can be used to automatically set a delay in the *System Thread's* loop function.

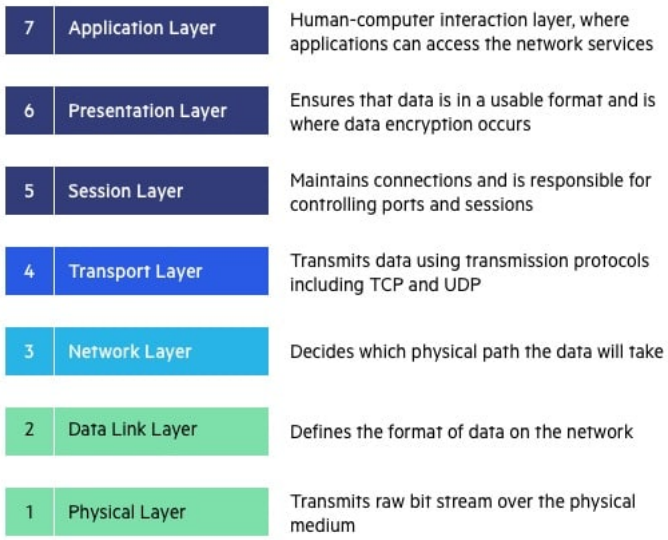
During development, the method `void println(const char* format, ...)` may be useful to transmit information to the serial console. The *System Thread* name is always prepended to the message to be printed.

Table @ lists the different *System Threads* used in Pollux III.

Source	Description	Priority	Stack
SupplyThread	Manages the power supplies	osPriorityNormal	2048
PowerThread	Manages the power monitors	osPriorityNormal	2048
WatchdogThread	Ensures that FreeRTOS never hangs	osPriorityNormal	2048
GUIThread	Manages the touchscreen	osPriorityNormal	2048
LoggingThread	Manages the logging on the storage	osPriorityNormal	2048
STMUARTDriver	Reads and writes data to Castor II	osPriorityRealtime	2048
Shell	Reads and writes to the serial console	osPriorityNormal	2048

4.1.3. Communication

Computer systems usually communicate with one another through the seven-stages Open System Interconnection (OSI) model.



All communication systems used in Pollux III are listed in the table @.

Application Layer	**Presentation Layer**	**Session Layer**	**Transport Layer**	**Network Layer**	**Data Link Layer**	**Physical Layer**
Supply control	MCP47 driver	-	24-bits payload	7-bits address	I2C frame	I2C4
Power monitor	INA239 driver	-	24-bits payload	3-bits multiplexer	SPI frame	SPI1
GUI	FT813 driver	-	8*n-bits flexible payload	End-to-end	QSPI pseudo-frame	SPI4
Castor interface	RoCo PowerBus	RoCo	RoCo	End-to-end	UART frame	UART2
Avionics interface	RoCo PowerBus	RoCo	RoCo	End-to-end	CAN-FD frame	CAN1
IPC	RoCo PowerBus	RoCo	RoCo	End-to-end	Memory	AXI

The Supply control, power monitor and GUI application layers represent the communication between Pollux III and onboard sensors/controllers. The Castor interface, Avionics interface and Inter-Process Communication, however, make use of the custom RoCo protocol and deserve a more thorough description in the following sections.

4.1.3.1. RoCo protocol

Abstract

RoCo, the Xplore's rover communication API, allows the Rover's different subsystems to communicate with each other. Every peer in a bus network can broadcast a message on the bus and this message might or might not be handled by the other peers.

Using a high degree of abstraction, this API ensures that a unique software can be shared across the subsystems without the need of porting or installing additional software. Since RoCo will also be used on embedded systems, the software is designed to have a low memory footprint and CPU overhead. Furthermore, static allocation predominates the code's architecture to avoid potential overflows.

In addition, RoCo also includes the communication protocol, that defines which packets will be transmitted and received through specific busses.

When using the RoCo API, please do only use the **RoCo.h** header file.

MessageBus API

RoCo's highest degree of abstraction is the MessageBus API, which defines how packets are defined, handled, sent or forwarded. This MessageBus class only requires the implementations to define standard read and write functions to be functional, as described later. Since memory allocation is static, it was necessary to define a maximum packet size (256 bytes), as well as a maximum number of unique senders (64). Moreover, there are at most 64 different packet identifiers.

Packet definition

Every MessageBus method requires an additional type info, which should define the structure of the considered packet. Packets should always be defined in accordance with the following scheme.

```
struct SomePacket {
    // Whatever content you need
} __attribute__((packed));
```

If you fail to define the structure as *packed*, extra padding bytes will be added to the good will of the compiler, which is the most undesirable effect possible since the code is supposed to be completely portable between different system architectures.

Another extra step to define a packet is to add it to the *protocol register* in *Protocol/ProtocolRegister.h*. Doing so will allocate memory for the template type of the structure previously defined.

```
#ifdef YOUR_PROTOCOL
REGISTER(SomePacket)
#endif /* YOUR_PROTOCOL */
```

Once those steps are done, you may use the MessageBus API to send this packet to any peer connected to the bus.

Define

```
template<typename T> bool define(uint8_t identifier)
```

The *define* method links the given packet structure to a unique identifier. Every identifier consist of two routing bits, which are reserved for now, and six identifier bits that can be freely assigned. It is recommended to define the different packets supported by the bus in the MessageBus' implementation's constructor. Once a packet is defined, it can be freely sent, received or forwarded. Otherwise, calling the later methods will result in failure. The *define* method will fail if one of the following conditions is met:

1. The packet identifier is already in use.
2. The packet size is too large (> 256 bytes).
3. The packet type is already defined with another identifier.

Send

```
template<typename T> bool send(T *message);
```

The *send* method allows the user to broadcast a defined packet on the bus. Note that thank to the template argument, it is not necessary to specify the nature of the packet being sent: The compiler will infer its nature automatically. The *send* method will fail if the provided packet has no assigned identifier.

Receive

```
template<typename T> bool handle(void (*handler)(uint8_t source, T*));
```

The *handle* method allows the user to receive a defined packet from the bus. This method requires the user to pass a callback function, that will handle the reception of the given packet. Depending on implementation, the source identifier might or might not accurately represent the sender of the packet. Note that thank to the template argument, it is not necessary to specify the nature of the packet being received: The compiler will infer its nature automatically. The *handle* method will fail if the provided packet has no assigned identifier.

Forward

```
template<typename T> bool forward(MessageBus* bus);
```

The *forward* method allows the user to redirect a defined packet from one bus to another. All packets having the same template type as the passed one will be

retransmitted to the given bus. The *forward* method will fail if the provided packet has no assigned identifier.

Writing an implementation of MessageBus

The *MessageBus* class defines two virtual methods that need to be implemented by subclasses:

```
virtual uint8_t append(uint8_t* buffer, uint32_t length) = 0; // Must be atomic
virtual void transmit() = 0;
```

As you may guess, the *append* method is used to send data to the bus controller. Note that in a multithreaded environment, it is required that *append* gets called in thread-exclusive context (usage of Lock/Mutex is recommended). In addition, *transmit* is used to flush eventual data that might be buffered in the bus controller.

Moreover, the implementation must ensure that *receive* is called whenever a buffer is received from the bus. The *receive* method has the following signature:

```
void receive(uint8_t senderID, uint8_t *pointer, uint32_t length);
```

By implementing the two above methods and calling *receive* when appropriate, the MessageBus implementation is complete and can be used on every platform that meets the dependency requirements.

IOBus: A buffered implementation of MessageBus

The IOBus implementation is designed to manage a classic pre-allocated buffer provided by the constructor. Transmission and reception is managed by a dedicated class called *IODriver*. The latter must be inherited and overwrite the following virtual methods:

```
virtual void receive(const std::function<void (uint8_t sender_id, uint8_t*
buffer, uint32_t length)> &receiver) = 0;
virtual void transmit(uint8_t* buffer, uint32_t length) = 0;
```

receive provides the *IODriver* implementation with a callback reception function. This callback function must be called whenever there is incoming data from the bus.

transmit signals the *IODriver* that the given data buffer must be sent to the bus. As for every I/O driver, it is required for the *transmit* method to have exclusive context.

NetworkBus: A TCP/IP implementation of MessageBus

The *NetworkBus* API inherits the IOBus implementation with a standard buffer size of 256 bytes. The only interest in analysing this implementation is the way the *IODriver* were developed. Note that *NetworkClientIO* and *NetworkServerIO* define slightly different implementations of the *IODriver*. For simplicity, we will assume the low-level implementations of these two *IODriver* are equivalent.

Using the <sys/socket.h> API, this implementation provides a simple way to communicate between two peers on a network. Note that this implementation is not fully compatible with the LwIP Stack, and thus cannot be used on embedded software.

To establish or interrupt a connection, two methods are available for each *IODriver*. Their usage is transparent.

NetworkClientIO

```
int8_t connectClient();
void disconnectClient();
```

NetworkServerIO

```
int8_t connectServer();
void disconnectServer();
```

For both implementations, the packet reception is done by reading from the socket through an independent thread. Transmission is simply done by writing to the TCP socket.

Protocol

The latest protocol (version 20W18) implements the following packet structures. Identifiers are not listed since they are bus-dependant.

Ping packet

Allows the user to assess the bus latency.

- Actual timestamp in nanoseconds as *std::chrono::time_point*

Connect packet

Signals a new connection on the bus.

- Actual timestamp in nanoseconds as *std::chrono::time_point*

Disconnect packet

Signals a disconnection on the bus.

- Actual timestamp in nanoseconds as *std::chrono::time_point*

Request packet

Signals the peers that a given resource is requested.

- Request UUID as *uint32_t*

- Action identifier as *uint8_t*
- Target identifier as *uint8_t*
- Request payload as *uint32_t*

Acknowledge packet

Acknowledges a request. Undefined if no request was previously performed.

- Request UUID as *uint32_t*
- Primary state of request as *uint8_t*

Response packet

Responds to a request. Undefined if no request was previously performed.

- Request UUID as *uint32_t*
- Action identifier as *uint8_t*
- Target identifier as *uint8_t*
- Response payload as *uint32_t*

Progress packet

Signals the requester that the resource is being processed and defines an actual state of the request. Undefined if no request was previously performed.

- Request UUID as *uint32_t*
- Progress information as *uint8_t*

Data packet

Broadcasts a generic data on the bus.

- Generic data as *uint32_t*

Message packet

Broadcasts a generic message on the bus.

- Message data as *uint8_t*[128]

Error packet

Signals the peers that the sender has experienced failure and defines an error identifier representing the situation.

- Error identifier as *uint8_t*

Building

Since RoCo is designed to be portable software, the same code base can be run on multiple platforms. However, some advanced features, such as Network I/O are not compatible with embedded systems. This is why there are several build configurations that need to be specified when targeting a specific platform.

Before including the *Roco.h* header file, it is necessary to declare which platform you are targetting:

- BUILD_FOR_CONTROL_STATION
- BUILD_FOR_NAVIGATION
- BUILD_FOR_AVIONICS
- BUILD_FOR_TESTING

It is also of crucial importance that all the peers on the same bus communicate with the same version of RoCo and, in particular, with the same protocol specification.

To enforce this, please do always specify and check the target protocol version in *Build/Build.h*. The latest implementation is version 20W18.

Examples

1. Client to client communication through a server.

Alice and *Bob* want to assess the quality of the link that connects them. In fact, *Alice* and *Bob* are both connected to *Carol*, which routes messages from *Alice* to *Bob* and from *Bob* to *Alice*. *Alice* tells *Bob* that she is going to send him her actual time. When *Bob* receives the message, he will compare his actual time to *Alice*'s actual time and compute by how much it differs. For simplicity, we assume *Alice*, *Bob* and *Carol* are one the same machine.

Since *Carol* is only supposed to send what she receives, using the *forward* method is appropriate. The implementation of the transmission and reception becomes trivial with the RoCo API:

```
void handle_ping(uint8_t sender_id, PingPacket* packet) {
    std::cout << "Ping C2C: " << (PingPacket().time - packet->time).count() <<
"ns" << std::endl;
}

int main() {
  NetworkServerIO* carol_io = new NetworkServerIO(42666);
  NetworkClientIO* alice_io = new NetworkClientIO("127.0.0.1", 42666);
  NetworkClientIO* bob_io = new NetworkClientIO("127.0.0.1", 42666);

  carol_io->connectServer();
  alice_io->connectClient();
  bob_io->connectClient();

  NetworkBus* carol_bus = new NetworkBus(charlie_io); // Alice-Carol-Bob bus
```

```
  NetworkBus* alice_bus = new NetworkBus(alice_io); // Alice-Carol bus
  NetworkBus* bob_bus = new NetworkBus(bob_io); // Carol-Bob bus

  carol_bus->forward<PingPacket>(carol_bus); // Rebroadcast on the Alice-Carol-
Bob bus
  bob_bus->handle(handle_ping); // Configure the reception

  PingPacket packet;
  alice_bus->send(&packet); // Send to Carol, who will send to Bob
}
```

4.1.3.2. Inter-thread communication

The RoCo protocol is used to allow different threads to communicate with one another.

4.1.3.3. Castor interface

4.1.3.4. Avionics interface

4.1.4. Supply management

4.1.5. Power monitoring

4.1.6. GUI

4.1.7. Logging

This section describes how the power system logs the mission data on a persistent storage. The logging system ensures that the mission data is robustly stored and can be later retrieved by the operator. As usual, the description of the design follows a top-down approach, first describing the high-level design and ending with the low-level persistent storage drivers.

The target flash memory is a 512Mb Micron Serial NOR Flash Memory (3.3V, Multiple I/O, 4KB, 32KB, 64KB, Sector Erase). To store data on this device, a three-layers design was implemented.

The first layer is implemented as a *System Thread* that gathers all information that must be logged on a persistent storage.

The middle layer - the so-called RocketFS (RFS) – provides the developers with a filesystem-like structure. It allows device mounting, unmounting, formatting, as well as file creation, deletion and streamed I/O operations.

The last layer consists of a low-level hardware-dependent code that allows read/write/erase operations on the flash memory.

4.1.7.1. Flash memory driver

4.1.7.1.1. Hardware definitions

The file ‘Inc/MT25QL128ABA.h’ contains definitions of the different commands that can be set in the flash control register to execute some operation. If it was decided, in the future, to change the flash memory (e.g., for higher capacity), this file should be modified accordingly. Please pay attention to the fact that we are using a NOR flash memory, meaning that, when a bit is erased, its value is one and when a bit is

programmed, its value becomes zero. Once a bit is zero, it cannot be reset to one unless a full erase of the current sector or subsector is performed.

4.1.7.1.2. QSPI wrapper

The file 'Src/qspi_driver.c' is used to provide an access to the HAL QuadSPI driver that is used to communicate with the flash memory. The reason to use an intermediate wrapper and not to directly use the HAL driver was to limitate the number of features available. In fact, the QSPI_CommandTypeDef construct gives the developer too much control over irrelevant flags or options. The wrapper is thus here to enhance the clarity of the code.

Since the QSPI_CommandTypeDef structure stores all information necessary to communicate to the flash driver, it was necessary to create a wrapping structure called 'Command' (io_driver.h). Nonetheless, it is deprecated to create this structure manually. Please use the Command **get_default_command**() function instead. If an address is required by the command, make use the **void with_address**(Command*, uint32_t) function and, if data is needed, use the **void with_data**(Command*, uint32_t) function.

Running a command is achieved through bool **qspi_run**(Command*, uint32_t). Further, data transmission and reception can be done with bool **qspi_transmit**(uint8_t*) and bool **qspi_receive**(uint8_t*) respectively. If polling is needed, for instance to wait for the flash memory to be ready to process to the next command, the function bool **qspi_poll**(Command*, uint32_t, uint8_t, bool) polls the given bit for an expected value. Note that all those functions return a boolean value representing whether the HAL API call was successful or not.

The implementation of those methods is pretty straight-forward once the reader is familiar with the documentation of the HAL QuadSPI [RD02] functions and the hardware specification of the flash memory [RD01].

4.1.7.1.3. I/O driver

Access to the flash driver is made through the following functions:

```
void flash_read(uint32_t address, uint8_t* buffer, uint32_t length)
void flash_write(uint32_t address, uint8_t* buffer, uint32_t length)
void flash_erase_all()
void flash_erase_subsector(uint32_t address)
void flash_erase_sector(uint32_t address)
```

First of all, reading from the flash memory is not very complicated once there is a QuadSPI wrapper. In fact, the only operations to do are to craft a command using the given address and data length. Then, the command is run and **qspi_receive**(uint8_t*) is used to indicate where to place the data read from the flash memory.

In contrast to the **void flash_read** function, writing is much more complicated. This is due to the fact that the write buffer is limited to 256 bytes (defined in Inc/MT25QL128ABA.h). Even the documentation of the flash memory does not state this limitation clearly. To mitigate this problem, the input buffer must be split in different sub-buffers of 256 bytes each. This is the reason why **void flash_write** contains a while loop alongside with some modulo operators: it has to split the buffer before attempting to write it to the flash memory. For each sub-buffer, **void __flash_write_page** does the job of writing the content of the buffer to the flash memory. Note that it is necessary to enable the write latch before executing the write command. At the end of the **void __flash_write_page** function, several

safety checks are made. Firstly, the microcontroller waits for the data to be completely transmitted (through polling) and then checks if a protection fault occurred while writing to the memory.

Once the reading and writing mechanism are fully understood, the erase operations are no surprise and won't be detailed in this document.

1.1.1 Concept

The RocketFS library has been implemented independently of all other software parts to avoid interference. For this purpose, a special github repository was created[1].

RocketFS is a standalone library, meaning that it does not require any additional software or hardware to be properly built or tested. In fact, as mentioned in the test procedure, a NOR flash memory emulator is available to test RocketFS.

The idea behind RocketFS was to find a lightweight solution to the issues caused by the FAT32 library, that led several boards to stop functioning during the main flight of the SA Cup 2019. Since lightweight does not mean simplistic, the most crucial features of a classic filesystem are still implemented. Among those are the ability to create and destroy independent files, a block management system, a memory protection mechanism and a streamed file access. To fulfil the need of saving the most important data, a feature to overwrite the oldest files when the device is full was also implemented. I will go through these features in the following sections.

1.1.1.1 Files

For a filesystem without files would not be called a filesystem, implementing those was a necessity. It is true that having a complete filesystem is not very useful if you think of the avionics software as a simple communicator that logs and transmits sensor data. For future avionics teams, however, it may be interesting to be able to store persistent data into a hostboard. This is the main reason why I implemented RocketFS. File creation, deletion and touching[2] are all implemented. Due to memory limitations, the current version of RocketFS limits the number of possible files to 16. Filenames are also limited to 16 characters.

1.1.1.2 Block management

Dynamic memory (such as a filesystem in our case) always comes with a way to split memory non-linearly in some blocks with a well-defined size. This is due to the fact that a file without a well-defined size must always store some metadata about where the file "segments" are located. To illustrate why such a feature is needed, let's suppose we are flying our beautiful rocket for the second time today. During the first flight, the avionics logged some state-related data inside a file called FSM_DATA. The second flight, however, is supposed to log only GPS coordinates in a GPS_DATA file. The avionics boots, creates a GPS_DATA file just after the last byte of FSM_DATA and starts filling it with data. The rocket takes off and touches down as expected (this time). At that point,

[1] https://github.com/EPFLRocketTeam/RocketFS

[2] No bad joke. Touching a file means setting the file's last modification date to now.

the ground station finds out that the touchdown state has not been detected during the first flight… it then sends a command to the avionics to write the missing data to the FSM_DATA file. The question is: where is this missing data going to be written?

A bad solution would be to write it just at the end of the FSM_DATA file because it would overwrite the beginning of GPS_DATA. It is now clear that a way to recognise which block is already used and which one is not, is needed.

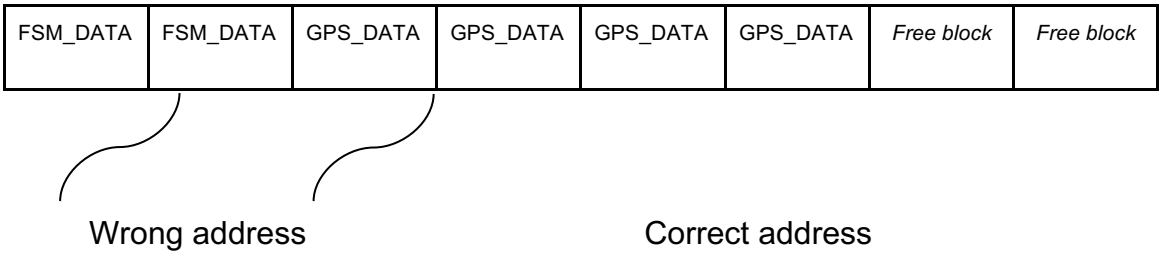


Fig. 1: Wrong addressing algorithm (linear) versus block addressing

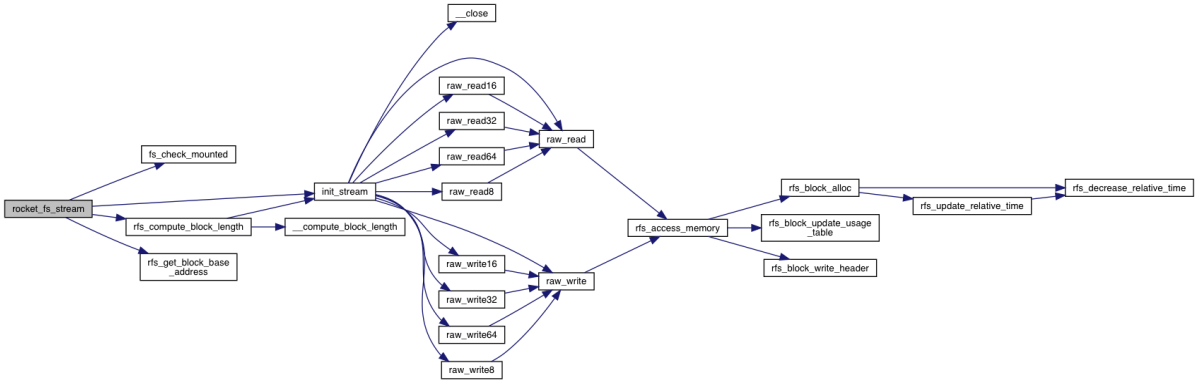
1.1.1.3 Memory protection

The memory protection feature has the only purpose of avoiding overwriting an existing file by error. If the Stream API is used, this feature guarantees that the read or write operations won't interfere with the other files.[3]

1.1.1.4 Streamed I/O

The Stream API was implemented, together with the memory protection feature, to prevent the developer from accessing the flash memory directly. Instead, passing by a stream ensures that read and write operations are always done to memory locations where they are permitted. Moreover, convenient functions are implemented to handle 16-bits, 32-bits and even 64-bits I/O operations. Another advantage of a streamed I/O is that the developer does not have to know at which address she or he is going to read or write. Since RocketFS and the flash driver upon which it relies are not designed to be thread-safe, it is only possible to create one stream at a time to deter the developer from accessing the library from different threads. This means that closing the stream at the end of using it is not only recommended, but it is also of greatest importance to avoid data corruption. These different design considerations are resumed in the following doxygen-generated figure.

[3] This is exactly the feature which failed during the Kaltbrunn 03/20 launch.



Base address resolution	Multi-byte I/O functions	Memory protection

Fig. 2: I/O Stream API layers

1.1.1.5 Resilience

Flight data is particularly sensitive. We cannot afford losing new data after the device's maximum capacity is reached. The concept behind resilience is to discard the oldest blocks in the filesystem to always ensure that the newest data is saved. The way to recognise which block is the oldest will be further explained in the 4.2.4 section.

1.1.2 Usage

All RocketFS functions require an instance of a FileSystem structure. The latter contains all information required to perform RocketFS I/O operations. In general, the library does not save state information. To allocate a FileSystem structure, do not use dynamic allocation.

Once a FileSystem instance is defined, the following functions are available:

```
void rocket_fs_debug(FileSystem* fs, void (*logger)(const char*));
void rocket_fs_device(FileSystem* fs, const char *id, uint32_t capacity, uint32_t block_size);

void rocket_fs_bind(
    FileSystem* fs,
    void (*read)(uint32_t, uint8_t*, uint32_t),
    void (*write)(uint32_t, uint8_t*, uint32_t),
    void (*erase_block)(uint32_t)
);

void rocket_fs_mount(FileSystem* fs)
void rocket_fs_unmount(FileSystem* fs)
void rocket_fs_format(FileSystem* fs)
void rocket_fs_flush(FileSystem* fs)
File* rocket_fs_newfile(FileSystem* fs, const char* name, FileType type)
void rocket_fs_delfile(FileSystem* fs, File* file)
File* rocket_fs_getfile(FileSystem* fs, const char* name)
void rocket_fs_touch(FileSystem* fs, File* file)
bool rocket_fs_stream(Stream* stream, FileSystem* fs, File* file, StreamMode mode)
```

Nonetheless, those functions are not always available and may have some runtime requirements. For instance, it is not possible to create a new file if the hardware I/O has not yet been bound to the FileSystem structure. As a consequence, there is a precise procedure to initialise and destroy the filesystem.

1.1.2.1 Device initialisation (required)

```
void rocket_fs_device(FileSystem* fs, const char *id, uint32_t capacity, uint32_t block_size)
```

Using the FileSystem instance, this function defines the hardware characteristics of the flash memory to which the device is bound. For optimal performance, ensure that the block_size equals the square-root of the capacity. Moreover, a device identifier may be declared. This identifier will be later available through the FileSystem structure and is not used by the library. Do not use this function while the filesystem is mounted.

1.1.2.2 I/O Bindings (required)

```
void rocket_fs_bind(
FileSystem* fs,
void (*read)(uint32_t, uint8_t*, uint32_t),
void (*write)(uint32_t, uint8_t*, uint32_t),
void (*erase_block)(uint32_t)
)
```

rocket_fs_bind is used to bind the filesystem instance to the hardware driver. The read, write and erase_block parameters are function pointers that are compatible with the FlashAPI specification. The read and write function pointers require an address, a buffer pointer and a length to be specified. The erase_block function pointer only requires an address as parameter.

1.1.2.3 Mounting (required)

```
void rocket_fs_mount(FileSystem* fs)
```

Mounting is an operation that loads the most important data stored in the device and makes it possible to create, modify and delete files. This function requires the device to be initialised and bound. If the device is corrupted or empty, the device is firstly formatted.

1.1.2.4 Unmounting (required)

```
void rocket_fs_unmount(FileSystem* fs)
```

This function saves the current state of the FileSystem structure into the device, that is which blocks have been (de)allocated. The filesystem must be mounted to perform this operation.

1.1.2.5 File management

```
File* rocket_fs_newfile(FileSystem* fs, const char* name, FileType type)
void rocket_fs_delfile(FileSystem* fs, File* file)
File* rocket_fs_getfile(FileSystem* fs, const char* name)
void rocket_fs_touch(FileSystem* fs, File* file)
```

The ability to create, delete and modify files is crucial to a filesystem. The way these functions are defined should be more or less intuitive. Nonetheless, the **rocket_fs_newfile** function takes a file type as parameter. Currently, only the RAW file type has been implemented.

Further, the File structure returned by **rocket_fs_newfile** and **rocket_fs_getfile** is used to create I/O streams and to retrieve some file metadata.

```
typedef struct File {
char filename[16];
uint32_t hash;
uint16_t first_block;
uint16_t last_block;
uint32_t length;
uint16_t used_blocks;
uint16_t reserved;
} File
```

As you may see on the structure definition, the length of a file and the number of blocks it uses can be found in this structure.

1.1.2.6 Stream API

```
bool rocket_fs_stream(Stream* stream, FileSystem* fs, File* file, StreamMode mode)
```

To use the Stream API, it is necessary to create a Stream instance beforehand. This structure will be the only way to read or write to a file. A stream may either be opened in APPEND or in OVERWRITE mode using the last parameter of **rocket_fs_stream**. The append mode is mainly used to write at the end of a file, without overwriting the previous content. This mode is not compatible with read streams. The overwrite mode, however, is mostly used for read streams. It sets the read cursor at the beginning of a file.

The Stream instance, once initialised by **rocket_fs_stream**, can be used to read or write to a file. The definition of this structure is as follows:

```
typedef struct Stream {
bool* eof;

void (*close)();

int32_t  (*read)(uint8_t* buffer, uint32_t length);
uint8_t  (*read8)();
uint16_t (*read16)();
uint32_t (*read32)();
uint64_t (*read64)();

void (*write)(uint8_t* buffer, uint32_t length);
void (*write8)(uint8_t data);
void (*write16)(uint16_t data);
void (*write32)(uint32_t data);
void (*write64)(uint64_t data);
} Stream
```

The generic read write operations are accessible through this structure. In addition to this, multibyte support allows to read or write standard *uint* without having to create dedicated buffers. The eof flag detects if the end-of-file has been reached. This flag should never be set in write mode. Note that the current implementation does not allow file sizes other than multiple of 64 bytes. Therefore, the read operation may read additional bytes, up to 63 bytes of 0xFF value, before setting the eof flag to one. One common error is to access eof as if it was a boolean. Namely, as it is a pointer, you should deference it before checking its value. For example, `while(!stream.eof)` is always false but `while(!stream->eof)` is the correct loop condition.
Most importantly, once a stream is not needed anymore, close it as soon as possible. In fact, remember that only one Stream is possible at a time.

1.1.2.7 Formatting

```
void rocket_fs_format(FileSystem* fs)
```

Formatting the filesystem restores it to its initial state. It erases all information about the allocated blocks and created files. Use this function with caution.

1.1.2.8 Flushing (optional)

```
void rocket_fs_flush(FileSystem* fs)
```

Flushing the filesystem from time to time is recommended. In the case of a power failure, flushing guarantees that the allocated blocks are correctly saved to the device and that no data is overwritten.

1.1.2.9 Debugging (optional)

```
void rocket_fs_debug(FileSystem* fs, void (*logger)(const char*))
```

Having information about the success or failure of a given operation is often very welcome. This is why a debugging feature is implemented in RocketFS. The second parameter of **rocket_fs_debug** is a function pointer to a logging function. This pointer is called with an information or error message as parameter.

1.1.3 Architecture

For RocketFS does not need to be tested with a hostboard, the github repository does not contain any additional file related to CubeMX nor any hardware-specific code. RocketFS compiles in the form of a static library. It is therefore necessary to link it correctly in the external application.

The directory structure is quite classic. The root directory contains an eclipse project file, a doxyfile (and its generated documentation), the main Src and Inc folders, as well as a Test directory[4]. It also contains a Headers folder and an example memory dump FLASH.DMP used by the flash emulator.

The Headers directory contains the only header files that should be included by external applications.

The main source code is dispatched between four source files, namely 'filesystem.c', 'file.c', 'stream.c' and 'block_management.c'. The most important structures are defined in the header files, such as the FileSystem, Stream and File structures.

File filesystem source file contains all functions related to high-level file management, such as file creation, file deletion, touching, device initialisation, mounting, etc. (see 4.2.2 for details).

'file.c' principally contains utility functions to handle file names. Among those are string hashing, string copying and string comparing.

The stream source file contains the multi-byte read and write function definitions. It also contains the current file read and write pointers.

On the other hand, 'block_management.c' along with 'block_management.h' should be only used internally by RocketFS. Do not attempt using these functions from an external application. This source file includes notably the way blocks are allocated, deallocated and reallocated, as well as the memory protection feature.

[4] See test procedure 2020_AV_TP_0008_AV_FLASH for further details.

1.1.4 Implementation

An overview of the current implementation of RocketFS will be given in this section. Please do not hesitate to refer to the real code and documentation available in the github repository. Not every aspect of the source code will be reflected here, yet only the most relevant and difficult parts of the code. Keep in mind that erasing a block costs time and that once a bit is set to zero, it is not possible to reset it to one, unless the whole block is erased. It is a property of NOR flash memories for which RocketFS was implemented.

1.1.4.1 Filesystem format

The device's memory is subdivided in different blocks. Some of those are used by the filesystem to store metadata, journals and backups.

Each data block starts with a block header of 16 bytes. The first four bytes represent a magic number (0xC0FFEE00) and is used to avoid overwriting of an already-allocated block if there is a bug. The next two bytes are the file identifier to which the given block is attached. Byte 6 to 7 represent the predecessor block ID. The next 8 bytes describe the used amount of memory inside the block. Each bit of these 8 bytes represent a cell inside a block. If a cell is written in the memory, its corresponding bit is set to zero. Since a block consists of 4096 bytes, there are 64 cells, each 64 bytes long. Moreover, if the block is the root of a file, 16 bytes are reserved to store the file name.

Block 0 is called the 'Core block' and is supposed to store a magic number and some metadata. The magic number is used to recognise if a device is compliant with the RocketFS format or if it can be fully erased. The way this magic number is implemented supports even high degrees of data corruption, meaning that even if the device is highly damaged, the filesystem will recognise that there is still some valid data and will not erase the device. Concrete implementation is shown in Appendix 1.

Block 1 is the 'Master partition table', containing information about the type of content (4 bits) and the age of a given block (4 bits). Since the size of a block is required to be the square root of the device's capacity, it is guaranteed that each block has its own entry in the partition table.

Block 2 is supposed to be used as a recovery partition table. However, this feature is not yet implemented.

Block 3 to 6 (inclusive) are also supposed to be backup partition table slots, but are not yet implemented as such.

Block 7 is the filesystem's journal. It should log each filesystem modification that is made to the filesystem. In the case of a power failure, it should be possible to restore a consistent state of the filesystem using the journal. Nevertheless, it is also still not implemented.

Block 8 to 4095 (inclusive) are data blocks. They store raw file data and have a particular structure.

Finally, blocks 4092 to 4095 (inclusive) are reserved for testing purposes.

1.1.4.2 Partition table

The partition table contains information about which block has been allocated and which one is free. If a block is allocated. Its allocation time and its file type are stored. Four bits represent the file type and the next four ones describe the age of the block. To avoid the need of erasing and reprogramming the partition table block each time a new block is allocated, two buffers are stored in the FileSystem structure: the partition table and its one-complement. When a file is closed or the file system is unmounted, the partition table is flushed to the device. If the above conditions are not met, data might be overwritten (Kaltbrunn 03/2020). It is worth noting that the partition table is not written 'as is' to the device. Instead, its complement is computed and written to the device. This has the advantage to increase the lifetime of a NOR flash memory since there are less zeroes to write and erase than if we wrote the partition table directly.

Even though the meaning of the file type nibble is quite clear, the age nibble is much less so. This information is used to estimate which block would be best to reallocate if the device's maximum capacity is reached. Time, in RocketFS, has a particular meaning and represents by no means an actual time or timestamp. In fact, the limitation of having to store a time information in 4 bits (value ranges from 0 to 15) forces us to find another time source. For this purpose, the age of the 'Core block' (block 0) is used. Initially, its value is set to 14 and each time the device loses 1/16 of its capacity, the value is decreased.

For instance, the BellaLui flash memory has 16MB of memory. The first allocated blocks have all an age value of 14. When there is only 15MB of memory left, the age value of all the first blocks is decreased to 13 and the new blocks are aged 14. When there is 14MB left, the first blocks' age are decreased to 12, the next blocks' age are decreased to 13 and the new blocks will be aged 14, and so on.

1.1.4.3 Mounting the filesystem

Mounting the filesystem is quite a complex procedure and will be explained here as an example to illustrate the methodology to develop RocketFS. To guide you, Fig. 3 shows the doxygen call graph generated for **rocket_fs_mount**.

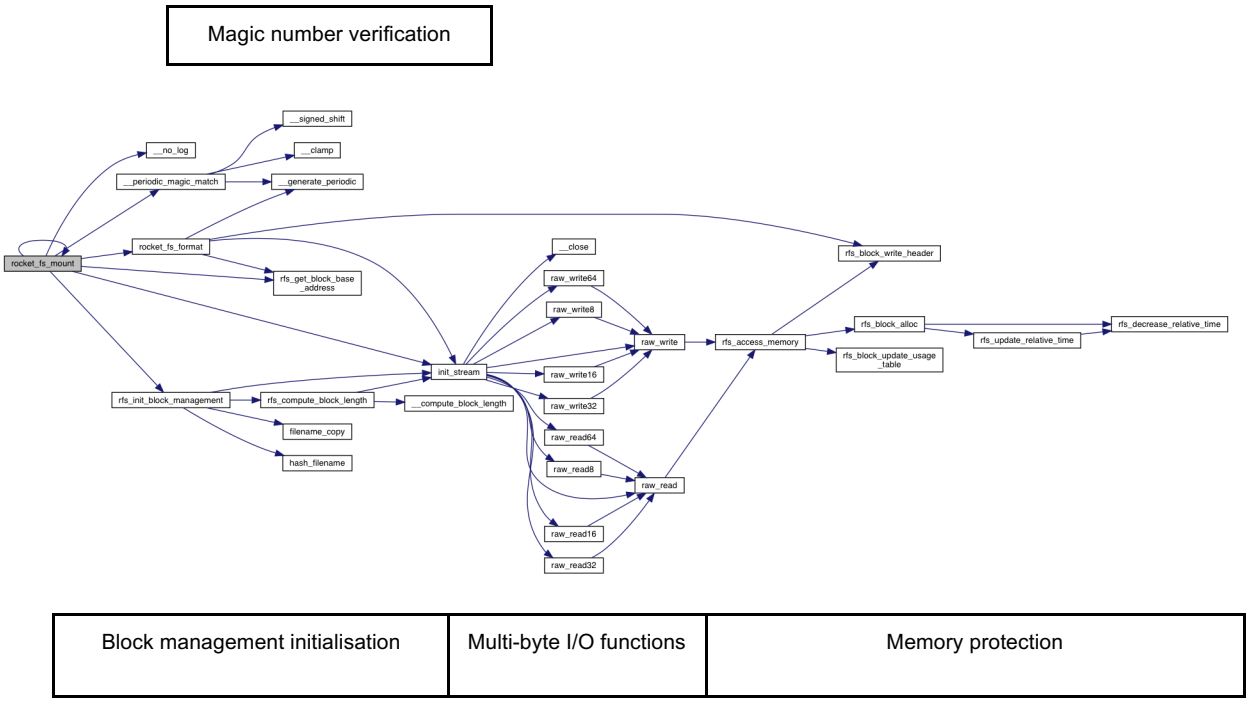


Fig. 3: rocket_fs_mount call graph

The first step in the mounting procedure is to compare the first bits of the device with the expected magic number value. If the magic number comparison algorithm (see Appendix) detects an incorrectly formatted device, it performs a call to **rocket_fs_format** to reset the device to an initial formatted state.

Otherwise, the partition table is read into the FileSystem structure's **reverse_partition_table** buffer and its complement is computed and stored in the **partition_table**. The block management has now to be initialised before the **mounted** flag is finally set to true.

This initialisation process happens in two stages. First, the partition table is scanned to detect all the created files in the device. Then, the block hierarchy is resolved and the file sizes computed.

Since flash memories are random access, it is not very performance-consuming to read the beginning of each block. This is actually the way the files are detected: the header of each allocated block is read and parsed. If the block has no predecessor, it must either be considered as the beginning of a file or as a lost block. The case of a lost block is a very special case when the device's maximal capacity is reached, and so won't

be explained here (refer to 4.2.4.5 Case 3 for details). On the other hand, if the block is recognised as the first block of a file, the actual filename is read, copied and hashed in a File structure. If the block has a predecessor, the **data_blocks** structure field is updated accordingly.

The second initialisation stage attempts to resolve the block hierarchy. Using the FileSystem's **data_blocks** substructure (must be thought of as a linked list), the size of the file is computed and updated in the correct File structure. It might be interesting to implement an algorithm to detect if there is a cycle in the linked list (caused by corruption), such as Tarjan's algorithm.

1.1.4.4 File management

The different files are stored in the FileSystem structure in the form of an array of File substructures. Since the number of entries in this array are very limited, RocketFS makes use of hashing to compute a file identifier associated with a file name. This guarantees O(1) file fetching. Note that hash collisions are handled by incrementing the hash value until there is no more collision. File names are limited to 16 characters (including or excluding the \0 character). The implementations of **rocket_fs_newfile**, **rocket_fs_delfile** and **rocket_fs_getfile** are, I believe, relatively clear.

1.1.4.5 Block allocation

Block allocation and memory protection are complementary to each other and are designed to be completely transparent to the external application. These features operate before each call to the native read and write functions through **rfs_access_memory**. This function is used exclusively by the Stream API. The purpose of **rfs_access_memory** is to detect when the read or write pointer are at the end of the current block and to jump to the correct read or write location. This location changes given certain circumstances that are listed below.

Case 1: If the requested address is inside the header protected area, the pointer is incremented to the first readable or writable byte.

Case 2: If the requested address is the last byte of a block, plus one, the end of a block is effectively reached and the requested operation must be analysed. If the Stream API wanted to read some bytes, then the function should return immediately and tell the Stream API to set the EOF flag to one. No bytes are remaining in the requested file.

Case 3: If the Stream API requested to write several bytes, it would be necessary to allocate a new block and update the file metadata accordingly. This procedure is achieved through **rfs_block_alloc**. This function iterates through the whole partition table in order to find the block with the lowest age value, which is the oldest block. If, during the iteration, a free block is found, then it is allocated and returned immediately. Otherwise, all blocks are allocated and

the device is full. This means that a block must be reallocated and the best candidate for this purpose is the oldest block allocated. However, reallocation breaks the block hierarchy (linked list) and there is a necessity to mitigate this issue. Thus, if the block to be reallocated has a successor, the latter is marked as a lost block, since its “predecessor” value doesn’t make sense anymore. During the initialisation of the block management, the lost blocks are reattached to the nearest preceding block that belongs to the same file. As you can imagine, this “reparation” can possibly fail and attach two blocks incorrectly. However, I don’t see any better way to recover a lost block that doesn’t require the rewrite of a full block. The only way to partially avoid this is either to ensure the device’s capacity is never reached (which we cannot guarantee) or to reduce the entropy of the filesystem: so please, avoid creating and deleting lots of files and reduce as much as possible the number of files you use.

This function may also restrict the number of bytes that can be read or written. Namely, no I/O operations should ever read or write for lengths greater than the length of the current block, minus the current pointer position. For this reason, an internal address is computed at the beginning of the function to represent the current pointer position (**internal_address = 1 + (*address - 1) % 4096**).

Finally, the usage table is updated if the requested operation is a write.

4.1.7.2. RocketFS

4.1.8. Watchdog

As a safety measure on the Pollux III hardware, a watchdog was implemented to keep track of FreeRTOS' state. If the FreeRTOS scheduler hangs because of a failing Thread or other reasons, the Pollux III software is automatically reset by the watchdog.

A watchdog is a particular timer in the low-power domain of a microcontroller. If the timer reaches its output compare value, an interrupt is triggered, and the Pollux III software is reset.

The purpose of the WatchdogThread *System Thread* is to reset this watchdog timer, so that it never reaches the output compare value in nominal conditions. If, however, the WatchdogThread hangs because of any other functionality on the board has failed, then the watchdog timer is not reset, an abnormal condition is detected, and the board is reset.

4.2. Castor II software

A 40-pin board to board extension connector allows the Pollux III hardware to be connected to an external device. Castor II was developed to transmit information gathered by Pollux III through a wireless connection and complies with the specifications of this 40-pins extension connector.
Castor II is a PCB featuring an ESP32 2.4GHz WiFi module, developed by Espressif. ESP-IDF V5.1 (Eclipse IDE variant) is used to develop the Castor II firmware.
The whole complexity of this software has been to embed an HTTP server, a Graphical User Interface (GUI) and a web API inside a microcontroller with very little RAM and flash memory.
The purpose of Castor II is to allow a computer or smartphone to connect to a WiFi network created by the ESP32. In this WiFi network, the web address http://192.168.1.42 opens a website generated by Castor II which displays information related to the power system. It allows a seamless control and monitoring of Pollux III's power supplies. In addition, an API is available under the web address http://192.168.1.42/api. See section 0 for further details on the usage of this API.
In the following sections, the end-user (who uses the website or the API) will be referred to as the **client**. The client performs requests (HTTP request) to Castor II, also known as the **server**. The server then responds (HTTP response) with HTML code that can be displayed using the client's web browser for example.
The hierarchy of the Castor II ESP-IDF project is decomposed as follows:

1. **System**: Contains the backend of the Castor II software. The folder itself contains the *System Thread* abstraction layer, consisting of *Thread.cpp/h* and *System.cpp/h,* as well the Event dispatcher and logic interfaces that connect the HTTP request handlers to the Pollux III data bus and the API. Its subfolders form a secondary hierarchy:

    a. **Status**: Provides some basic feedback on the state of Castor II through the PCB's internal LEDs and the serial console.

    b. **Telemetry**: Manages the API backend.

2. **HTTP**: Takes the responsibility to handle HTTP requests and to deliver the raw HTML data to the client.

    a. **UI**: Handles all HTTP requests related to the graphical user interface itself (the multiple pages on the website).

    b. **Common**: Handles all HTTP requests related to auxiliary libraries, scripts, and graphical elements needed (Bootstrap 4, JQuery, the Xplore logo, the API, the main JS script).

    c. **Data**: Contains the raw HTML data that is transmitted through the HTTP handlers in UI and Common.

3. **Drivers**: Encompasses the basic drivers to allow communication between Castor II and Pollux III through the 30-pin interface connector.

4. **RoCo**: Contains the RoCo communication protocol stack. See section 4.1.3.1 for further details.

4.2.1. Website

Each time the client wants to reach a webpage, it uses its web browser to perform an HTTP request to the corresponding website. It is useful to firstly understand how HTTP requests are performed:

When browsing to an URL such as https://epfl-xplore.ch/kerby-project-2023/, the following analysis is made by the web browser:

The DNS address must be first used to find the IP address of the target server. The IP address contains information on how to reach the server that stores the website. Here, `epfl-xplore.ch` is the DNS address which can be resolved by a DNS lookup to the IP address `87.98.255.4`.

The port defines to what TCP server socket the client refers to on the target server. It is often uniquely defined by the protocol being used. Here, `https://` defines the protocol used (HTTPS) and the port used (443).

The URI defines what resource is requested by the client. An URI is a relative link to the website's IP/DNS address. `/kerby-project-2023` is the URI requested by the client on the HTTP server.

4.2.1.1. Access point

The WiFi access point created by the ESP32 is configured in Main.cpp. It setups the WiFi network name, password and related configurations. Table @ lists the configuration used:

Configuration entry	Value
SSID	ERC_Xplore_2.4GHz
Password	*See section 5.3.11*
Channel	11
Maximum connections	4
Authentication mode	WPA2/PSK
IP address	192.168.1.42
Gateway address	192.168.1.1
Subnet mask	255.255.255.0
Bandwidth	20MHz

4.2.1.2. HTTP server

The HTTP server in HTTPServer.cpp/h allows the main code in System.cpp/h to register the multiple URIs needed by the website and handles the client's requests by serving them with the corresponding content.

The Castor website consists of multiple interlinked URIs.

An HTTP request handler in HTTPRequestHandler.cpp/h is associated with each of these URIs.

The website structure is summarized by the following tables.

URIs related to the frontend:

URI	Content-type	Source files	Description
/	text/html	Dashboard.cpp/h	Gives a system overview
/supplies	text/html	Supplies.cpp/h	Shows the state of the supplies
/safety	text/html	Safety.cpp/h	Shows safety information

/diagnostics	text/html	Diagnostics.cpp/h	Shows diagnostics information

Real webpages are associated with these four URIs. They define the static graphical content the end-user sees on its web browser. Hyperlinks embedded on the webpages connect the pages with one another. The webpages include the backend scripts and libraries needed to make the website dynamic and show real-time information about the power system.

URIs related to the backend:

URI	Content-type	Source files	Description
/api	application/json	Backend.cpp/h	API
/js/main.js	application/javascript	MainScript.cpp/h	Binds API to webpages
/download	application/octet-stream	Download.cpp/h	Transfers mission data

A Javascript script /js/*main.js* performs the API calls to the */api* URI that make the website dynamic either on a regular basis (to refresh the sensor values) or momentarily when the user interacts with the GUI (to perform a specific action). The */download* URI is used to download the mission data stored on Pollux' persistent flash memory.

URIs related to the libraries:

URI	Content-type	Source files
/css/bootstrap.min.css	text/css	Bootstrap4CSS.cpp/h
/js/bootstrap.min.js	application/javascript	Bootstrap4JS.cpp/h
/js/jquery.min.js	application/javascript	JQuery.cpp/h

The Bootstrap 4 library is used to add a modern look to the HTML webpages. It mainly uses CSS and Javascript to add graphical content and animations to the website.
JQuery is a well-known Javascript library that allows the HTML webpages to be updated dynamically. It is used to change the website's content when new sensor data is available from Pollux and to perform specific actions when the end-user interacts with the website.

4.2.2. Thread management

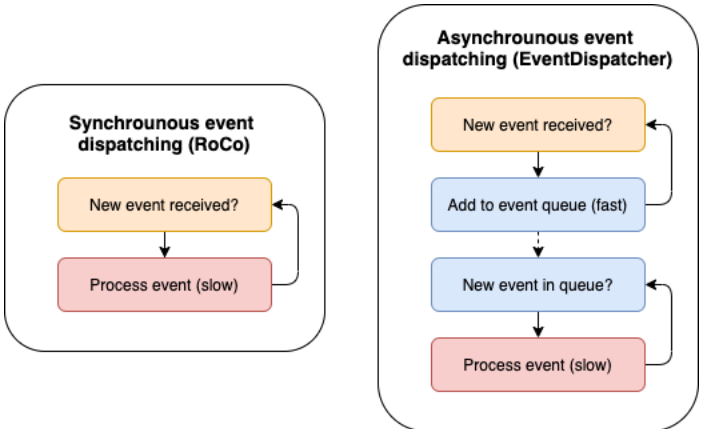
To simplify the development of the Castor II firmware, the *System Thread* abstraction layer developed for Pollux III was ported to the ESP32 hardware platform. See section 4.1.2.2 for further details.
The main difference in implementation between Pollux III and Castor II is that the ESP32's entry point for user code is written in C++ already, eliminating the need for a `void systemd_init()` function. Instead, the thread constructors are called by the constructor *System::System* class in *System.cpp*.

The following API changes between the STM32 port and the ESP32 port had to be performed:

- `void Thread::start()` must now be called to start the *System Thread* and add it to the FreeRTOS scheduler's task list.
- `void Thread::terminate()` is no longer available.

4.2.3. Event dispatcher

In Pollux III, the inter-thread communication was guaranteed by the RoCo protocol. By design, RoCo is synchronous and does never make a copy the information to be sent to a bus. This implies an obvious memory efficiency and speed advantage.
Nevertheless, when using a *LoopbackDriver*, the *System Thread* that sends a message is also responsible for calling the handlers registered for this message.
For example, if the user interface *System Thread* wants to send a signal to stop a power supply, it sends a message to the RoCo bus which calls the handler that should stop the power supply. The problem here is the user interface *System Thread* must be blocked until the handler is done stopping the power supply.
This is not acceptable in event-driven applications and with this regard, an Event dispatcher was implemented and is used to carry small signals/events through the Castor II firmware.
The event dispatcher concept presented here is grossly inspired by the Javascript DOM and Java AWT. Figure @ depicts the difference between synchronous event dispatching and asynchronous event dispatching. With synchronous dispatching, the speed and latency of the *System Thread* is always limited by the speed at which each event is processed. In asynchronous dispatching however, every time a new event is received, it is added to an internal queue. Another *System Thread* then addresses the event in parallel and bears the load of processing the event.



The event dispatcher has two methods to fulfil its functionality.
Firstly,

```
bool setUpdateSubscriber(subscriber_id, std::function<void(UpdateEvent*)>)
```

is used to register an event handler for a given subscriber ID. The subscribers IDs must be one of *EVENT_LOG*, *FSM*, *SUPPLY_MGR*, *HEALTH_MGR*, *USER_FB* in order of dispatch priority. The function given in argument to this method will be called every time an event is requested. All events are then broadcasted to the subscriber.
This method returns false if there is a subscriber already registered for the given subscriber ID or if the subscriber ID is incorrect.

Secondly,

```
bool requestEvent(UpdateEvent);
```

is used to add an event to the event queue. Custom defined events in *Events.h* can be defined by inheriting the *UpdateEvent* class. An event type must be assigned to the custom event by calling the *UpdateEvent* constructor with a given *event_t* type. These types are currently defined by the following enum in *EventDispatcher.h*:

```
typedef enum {
  INVALID_TYPE,
  READY,
  USER_CONNECTED,
  USER_DISCONNECTED,
  SELFTEST,
  SUPPLY_START,
  SUPPLY_STOP,
  FAULT,
  CRASH
} event_t;
```

This method returns false if the event queue or the given event is invalid is full.

Implementation Note. The event queue is implemented as a circular buffer. Elements from the queue are accessed by applying the modulo operator:

```
  event_queue[index % EVENT_QUEUE];
```

Two counters are used to queue track of the event queue's state: a master index and a slave index. The master index is incremented every time a new event is added to the event queue. The slave index is incremented every time a new event is processed from the event queue. While the slave index is smaller than the master index, it means that the event loop must process new events:

```
  while(slave_index != master_index) {
    UpdateEvent* event = &event_queue[slave_index];
    dispatchEvent(event);
    slave_index = (slave_index + 1) % EVENT_QUEUE;
  }
```

Since multiple *System Threads* can add events to the event loop, it is necessary to suspend the FreeRTOS scheduler while an event is being added to the queue:

```
  vTaskSuspendAll();

  uint8_t next_index = (master_index + 1) % EVENT_QUEUE;
  event_queue[master_index] = event;
  master_index = next_index;
xTaskResumeAll();
```

This avoids race conditions between multiple *SystemThreads*.

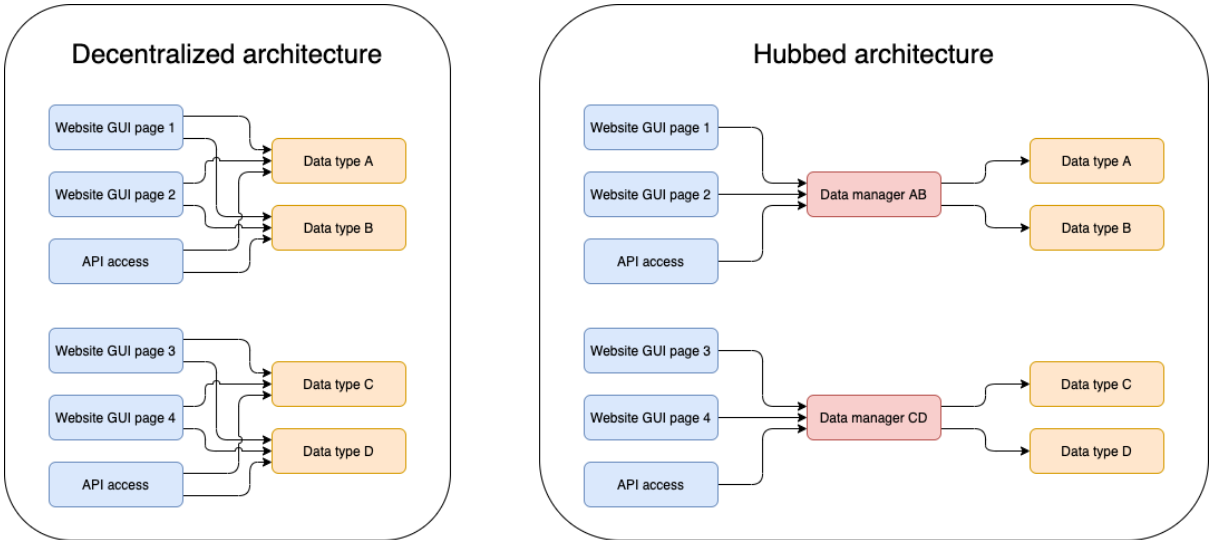
Improvement Note. The event dispatcher implementation is suboptimal. It could be improved by using FreeRTOS event groups for lightweight events or FreeRTOS queues for more complex events.

4.2.4. Data management

The information flow between the HTTP server and the Pollux III sensors is quite complex. The code readability was improved by using a design pattern involving three entities:

- The data supplier (e.g., sensor data, sensor state, saved mission data)
- The data manager
- The data consumer (e.g., website, API)

Figure @ illustrates how the usage of a three-entity design pattern simplifies the software architecture compared to a decentralized architecture. The data suppliers are coloured in orange, the data manager in red and the data consumers in blue.



A data manager serves as a "hub" between different data sources from Pollux III and stores the data deemed important to display on the website. For instance, it computes statistics over the power supplies' data, such as the average power or the voltage ripple on a given voltage rail and allows the client to access this data from the website or the API.

Each data manager is a *System Thread*.

4.2.4.1. Sensor data

The *SensorManager* class in *SensorManager.cpp/h* handles the *Power_BusInfo* RoCo packet. This packet contains sensor data related to a power supply's voltage rail and is regularly transmitted by Pollux III.

The packet data is used to fill the data structure holding all necessary information related to a voltage rail:

```
typedef struct {
  float voltage;
  float current;
  float energy;
  float temperature;

  float last_voltages[SAMPLE_MEMORY];
  uint32_t last_voltages_index;

  float last_currents[SAMPLE_MEMORY];
  uint32_t last_currents_index;
```

```
} sensor_info_t;
```
In this data structure, the latest voltage, current, energy and temperature readings are recorded. In addition, two circular buffers of size `SAMPLE_MEMORY` hold the latest voltages and currents recorded for each voltage rail.
Using the circular buffers, the SensorManager class allows the computation of statistics on a given power supply through the following methods:

```
float computeBatteryCharge();
float computeAveragePower();
float computeVoltageTransient(sensor_t sensor);
float computeCurrentTransient(sensor_t sensor);
float computeVoltageRipple(sensor_t sensor);
float computeCurrentRipple(sensor_t sensor);
```
The latest sensor data can also be accessed through:

```
sensor_info_t getSensor(sensor_t sensor);
```

These functions are used by the API in *SensorsTelemetry.cpp/h*.

Implementation Note. The statistics for the power supplies were computed according to the following considerations:

- The battery charge estimator computes the average battery voltage over the sample memory and computes the average battery cell voltage by dividing by the number of cells in series (7). Discharge tests of the battery pack were performed using an active load and the discharge profile of a Sony Murata VTC6 cell could be extracted from this experiment. This discharge profile is stored in `cell_profile_x[]` for the voltage in volt and in `cell_profile_y[]` for the remaining charge in percent. A Lagrange interpolation is then performed to obtain the actual state-of-charge of the battery pack.
- The average power is computed by averaging the power readings over the sample memory.
- The voltage and current transients are calculated by computing the average readings over the sample memory and finding the largest value with respect to the average.
- The voltage and current ripples are estimated by the standard deviation of the readings over the sample memory.

Improvement Note. The battery charge estimator could make use of a Kalman filter or similar estimation methods to obtain the state-of-charge not only from the battery pack voltage but also from the embedded energy meter.

4.2.4.2. Health data

The *HealthManager* class in *HealthManager.cpp/h* provides the API with information regarding the health of the power system. The API can request the *HealthManager* to reset the Pollux III software or the Castor II software.

Additionally, the *HealthManager* implements a self-test functionality for future developments on the 2024 digital twin project.

The management of a given controller is managed by the `controller_t` enum. Its value can be one of *`SUPERVISOR`*, *`CTA`*, *`CTB`*, where *SUPERVISOR* represents Castor II, *CTA* represents Pollux III and *CTB* is a legacy redundant controller from Pollux II, which is kept for backward compatibility.

```
void reset(controller_t ct);
void selftest(controller_t ct);
```

`reset` and **`selftest`** are the actions that the API can call. The *HealthManager* can reset Castor II or Pollux III but a self-test can only be performed on Pollux III for now.

```
controller_info_t getInfo(controller_t controller);
```

`getInfo` retrieves the current health data. The available data is stored in the `controller_info_t` data structure:

```
typedef struct {
 uint8_t state;
 uint64_t ping;
 uint64_t last_update;
 float heap;
 float flash;
} controller_info_t;
```

The fields of this data structure are defined as follows:

- The `state` is propagated to the *HealthManager* through the *EventDispatcher* for Castor II and through the RoCo PowerBus *Power_ControllerHealth* packet for Pollux III. The state can be one of:
  - **`STATE_RESET`**: Default state.
  - **`STATE_BOOTING`**: State set when the microcontroller starts.
  - **`STATE_READY`**: State when the microcontroller is functional.
  - **`STATE_SYNC`** (Only for Castor II): State when a user is connected to the WiFi network.
  - **`STATE_CRASH`**: State when an unrecoverable error has occurred.
- The `ping` is a measure of the round-trip latency between two microcontrollers. It is zero for Castor II and the RoCo PowerBus *PingPacket* is used to compute the round-trip time between Castor II and Pollux III.
- The `last_update` is the time at which the health data was last updated. It is timestamped upon reception of the *PingPacket* in the case of Pollux III and timestamped at each loop of the *HealthManager System Thread*.
- The `heap` field represents the heap memory usage in percent, as computed through FreeRTOS in both Pollux III and Castor II. The RoCo PowerBus *Power_ControllerHealth* packet is used to transmit this information from Pollux to Castor.

- The `flash` field contains the external flash memory (logging storage) usage in percent for Pollux III. It is zero for Castor II, as no data logged directly in Castor. The RoCo PowerBus *Power_ControllerHealth* packet is used to transmit this information from Pollux to Castor.

To perform a self-test on Pollux III, the API must call **`selftest`**. Afterwards, **`getSelftestInfo`** can be used to get the current progress of the self-test and potential errors that may have occurred.

```
selftest_info_t getSelftestInfo(controller_t controller);
```

The current self-test progress information is stored in the `controller_info_t` data structure:

```
typedef struct {
 bool completed;
 uint64_t start_time;
 uint64_t end_time;
 uint8_t progress;
 uint8_t faults[16];
 uint8_t fault_index;
} selftest_info_t;
```

The RoCo PowerBus RequestPacket is used to request a self-test from Pollux III. Pollux then periodically transmit a *ProgressPacket* to indicate what is currently tested and the `progress` field is updated. When errors are encountered during the tests, several *ErrorPacket* are transmitted: the `faults` and `fault_index` fields are updated. Once the self-test is finished, a *ResponsePacket* is transmitted: `completed` and `end_time` are then set accordingly.

4.2.4.3. Supply state data

The SupplyManager class in SupplyManager.cpp/h allows the API to retrieve the current state of the power supplies, i.e., whether they are enabled or disabled.
The following structure stores the power supplies' state.

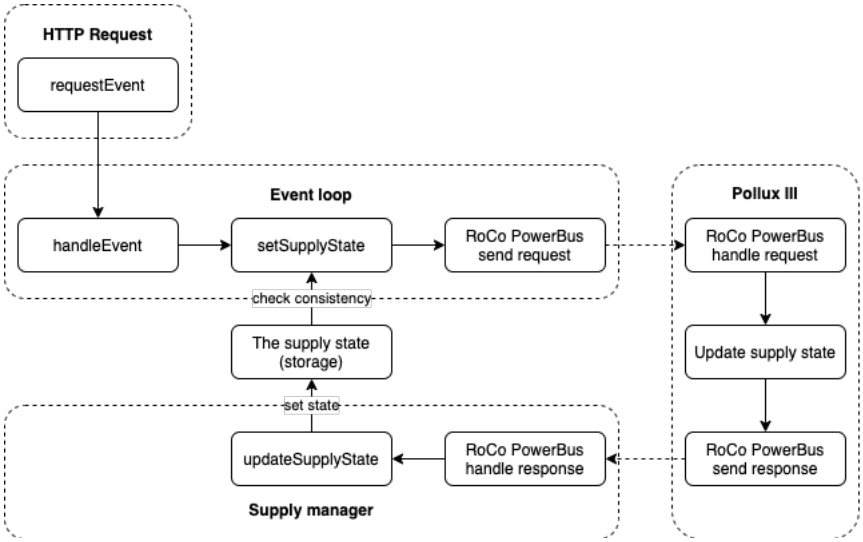
```
typedef struct {
 bool running;
 bool cta_running;
 bool ctb_running;
 int64_t uptime;
 int64_t last_event;
} supply_info_t;
```

The API performs event requests of type **`SUPPLY_START`** or **`SUPPLY_STOP`** to the EventDispatcher. The fields `running` and `last_event` are updated once the event is executed on the event loop. The power supply's `uptime` is also computed and updated in the structure. Then, a RoCo PowerBus *RequestPacket* is sent to Pollux III. Once Pollux has effectively changed the power supply's state, it answers the *RequestPacket* by sending a *ResponsePacket*, acknowledging the successful completion of the state change. At this point, `cta_running` now contains the power supply's state according to Pollux. Note that `ctb_running` is a deprecated field kept for backward compatibility.

The API can access the state of the power supplies and the uptime statistics through the following methods:

```
bool isRunning(supply_t device);
bool isRunningForCTA(supply_t device);
bool isRunningForCTB(supply_t device);
float getUptime(supply_t device);
```

Figure @ summarizes the SupplyManager software architecture.



Implementation Note. For a given power supply, its uptime in the `supply_info_t` data structure is only updated when its state changes. The real-time uptime is computed by adding the uptime already taken into account (in `supply_info_t`) and add the additional uptime between the last event and the current time (remember that `running` has a value of either zero or one):

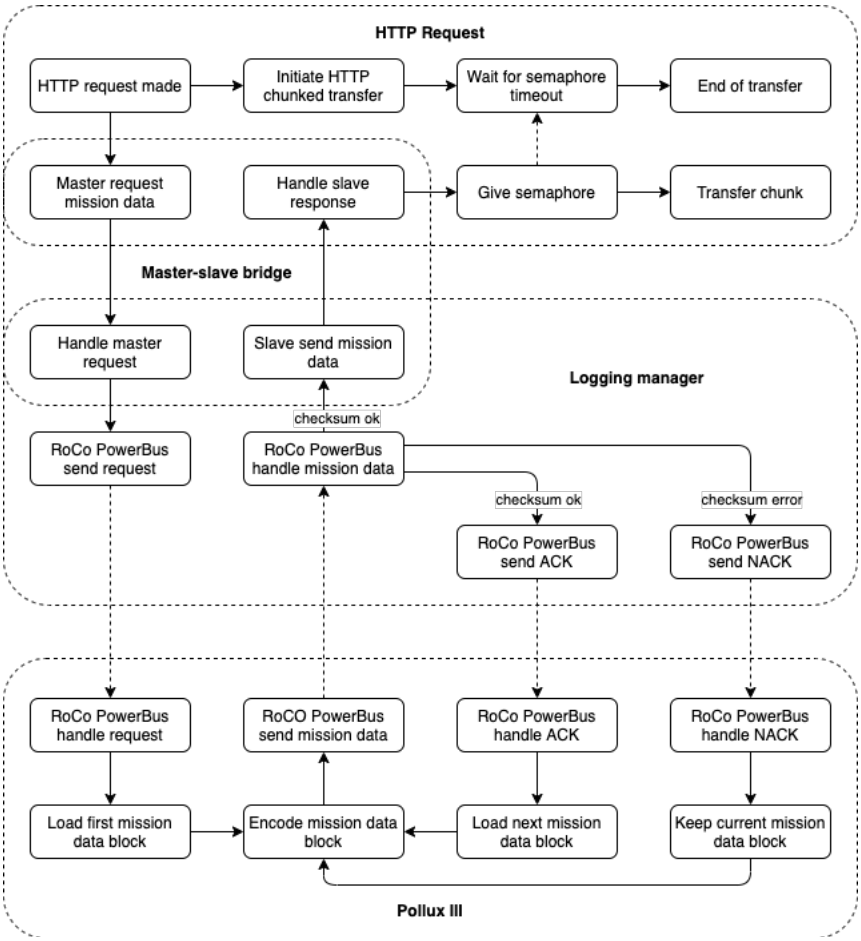
```
uptime + running * (time - last_event);
```

4.2.4.4. Logging data

The *LoggingManager* class in *LoggingManager.cpp/h* handles the transfer of mission data stored in Pollux' flash memory to the client. See section 4.1.7 to obtain a detailed description of the mission data transfer from Pollux' point of view.

The storage and transfer of mission data is by far the most complex piece of software in Castor/Pollux.

The software architecture is similar to the one used for the SupplyManager in section 4.2.4.3. Nevertheless, an additional interface layer manages the data transfer between the HTTP Request and the Logging manager. This layer is called the Master-Slave Bridge and prevents recursive dependencies between the HTTP Request and the Logging manager (See improvement note). The full architecture is depicted in figure @.



Every time Pollux starts, a new mission ID is generated. This ID is a unique number representing the current mission. The API can obtain the current mission ID by the method:

```
uint32_t getMissionID(bool cta);
```

The `cta` parameter is unimportant and is only kept for backward compatibility with Pollux II. This function sends a *RequestPacket* to the RoCo PowerBus to achieve its goal.

The client can therefore download the data from a mission by performing an HTTP GET request to http://192.168.1.42/download with the *m* GET field set to the requested mission ID.

From there on, the mission data flow is managed by 4 entities:

1) The HTTP request handler in *Download.cpp/h.*
2) The Master-Slave bridge in *Download.h*.
3) The logging data manager in LoggingManager.cpp/h.
4) Pollux III through the RoCo PowerBus.

The HTTP request handler on the server side accepts the request and provides the mission data through the following steps:

1) The transfer is initiated by setting the HTTP response headers. The content-type is set to `application/octet-stream` and the content-disposition is set to `attachment`.
2) The Master-Slave Bridge is called to initiate the corresponding transfer of the mission data.
3) The HTTP request handler waits for the transfer to finish (see implementation note).
4) The HTTP status 200 OK is sent to the client.

5) The HTTP connection is closed.

The Master-Slave bridge simply connects the HTTP request handler to the logging data manager.

The following protocol covers the mission data transfer between Castor and Pollux:

1) When the logging manager receives the download request from the Master-Slave Bridge, it forwards the request to the RoCo PowerBus through a *RequestPacket.*

2) On Pollux, the first mission data block corresponding to the given mission ID is loaded.

3) Pollux transmits the current mission data block to the RoCo PowerBus through a *PayloadPacket*, which can transport a buffer up to 512 bytes. A CRC16 checksum of the packet content is computed and appended to the packet.

4) Upon reception of the PayloadPacket, the logging manager checks whether the data is correct by computing the CRC16 value of the packet and comparing it to the ground-truth CRC16 value already appended to the packet.

5) If the payload is valid, Castor sends a *ResponsePacket* with an ACK target to Pollux, confirming good reception of the packet.

6) Pollux then loads the next mission data block and performs step (3).

7) If the payload is invalid, Castor sends a *ResponsePacket* with a NACK target to Pollux, implying that a communication error occurred in the last payload transfer.

8) Pollux reloads the same mission data block and performs step (3).

9) A payload of length zero sent by Pollux indicates an end-of-transfer.

This concludes the standard procedure to download the data from a mission.

The API can also request a full erase of the logging memory on Pollux. By design, the flash memory on Pollux can only store around 1h of data, which motivates the need of clearing this memory regularly. This feature is achieved by the following method:

```
void eraseFlashMemory(bool cta);
```

The `cta` parameter is unimportant and is only kept for backward compatibility with Pollux II. This function sends a *RequestPacket* to the RoCo PowerBus to achieve its goal.

Implementation Note. When transmitting large amounts of information, such as mission data, there is the possibility that a digital communication error occurs between Castor and Pollux. Even though CRC16 verification is implemented for verifying the integrity of a packet, it does not guarantee its correctness, nor does it guarantee that all transmitted packets will be correctly detected (the packet ID might be corrupted).

This is particularly an issue because Castor must acknowledge every payload packet it receives, so that Pollux sends the next payload. Thus, if a payload packet or an acknowledge is lost, the transfer fails completely.

To avoid that the HTTPRequestHandler stalls in such a condition, a semaphore with a limited timeout is used. If the transfer fails, the semaphore will timeout and the partially transferred mission data will still be correctly delivered to the client.
Improvement Note. The Master-Slave Bridge is an additional layer of complexity that does not necessarily improve the code readability. Getting rid of this layer should be possible but I didn't manage to do it because the g++ compiler complained about some sort of circular dependencies around the LoggingManager and the corresponding HTTPRequestHandler…

4.2.5. API

4.2.5.1. API specification

The Castor API allows the client to perform requests to the power system. It is accessible through HTTP GET requests under the address http://192.168.1.42/api, when connected to the Castor network. The API always send responses to the client in the `application/json` content-type. The responses are compliant with the IETF RFC 8259 JSON standard.
If the syntax of a request is correct, Castor responds with an *HTTP 200 OK* status. Otherwise, an *HTTP 400 Bad Request* error is sent.
The GET request uses the following fields to perform a valid request.

Field	Name	Description
a	Action	Action to be performed. Must be one of: *exe*, *get*, *set*.
t	Target	Targeted resource on Castor.
v	Value	Payload to the targeted resource. Only used if the action is *set* or *exe*.

The actions to perform on Castor and Pollux are divided into three categories: executers, getters, and setters.
For each of these action categories, the following API calls are implemented:

4.2.5.1.1. Executer requests

Target	Payload	Description
start	Supply ID	Start the given power supply.
stop	Supply ID	Targeted resource on Castor.
reset	Controller ID	Resets the targeted controller.
test	Controller ID	Starts a self-test on the targeted controller.
erase	Controller ID	Erases the logging flash memory of the targeted controller.

Note: Supply ID must be one of: 5v, 15v, 24v, 48v, corresponding to lva, lvb, hva, hvb.
Note: Controller ID must be one of: cta, supervisor, corresponding to Pollux and Castor.

4.2.5.1.2. Executer responses

A single standard response is sent back to the client.

JSON field	Type	Description
success	boolean	True if the operation succeeded
message	string	User-friendly operation feedback.

4.2.5.1.3. Getter requests

Target	Description
supplies	Returns the state of the power supplies.
sensors	Returns the latest sensor measurements and statistics.
diagnostics	Returns the latest system diagnostics.
temperatures	Returns the latest measured temperatures.
transients	Returns the latest transients that have occurred on the supplies.

supervisor_test	Returns the self-test results of the supervisor controller (Castor).
cta_test	Returns the self-test results of the CTA controller (Pollux).
mission_id	Returns the current mission ID.

4.2.5.1.4. Getter responses

The standard JSON field **success** and **message** are sent back to the client, alongside with custom JSON fields for each possible target.

For all targets:

JSON field	Type	Description
success	boolean	True if the operation succeeded
message	string	User-friendly operation feedback.

For **supplies** target:

JSON field	Type	Description
b5V	object	JSON description (see below) of the state of the LVA bus.
b15V	object	JSON description (see below) of the state of the LVB bus.
b24V	object	JSON description (see below) of the state of the HVA bus.
b48V	object	JSON description (see below) of the state of the HVB bus.

Supply state object:

JSON field	Type	Description
running	boolean	True if the supply is enabled.
uptime	number	Proportion of time enabled over total time.

For **sensors** target:

JSON field	Type	Description
battery	object	JSON description of the readings on the input bus.
motors	object	JSON description of the readings on the direct output bus.
b5V	object	JSON description of the readings on the LVA bus.
b15V	object	JSON description of the readings on the LVB bus.
b24V	object	JSON description of the readings on the HVA bus.
b48V	object	JSON description of the readings on the HVB bus.

For **battery** JSON field:

JSON field	Type	Description
charge	number	Battery charge left on the power system.
runtime	number	Estimated battery time left at the same power.
voltage	number	Input bus voltage.
production	number	Power incoming in the power system.
current	number	Current incoming in the power system.
ripple	number	Voltage ripple on the input bus.

For all other JSON fields:

JSON field	Type	Description
voltage	number	Bus voltage.
consumption	number	Power consumed.
current	number	Current consumed.
ripple	number	Voltage ripple on the bus.

For **diagnostics** target:

JSON field	Type	Description
supervisor	object	JSON description (cf. below) of the Castor's diagnostics.
cta	object	JSON description (cf. below) of the Pollux' diagnostics.
ctb	object	Kept for backward compatibility.

Controller diagnostics object:

JSON field	Type	Description
state	string	State of the controller.
ping	number	Round-trip time between the controller and Castor
last_update	number	Last timestamp the values above were updated.
heap	number	Proportion of heap memory available.
flash	number	Proportion of logging flash memory available.

For **temperatures** target:

JSON field	Type	Description
battery	number	Temperature on the input bus.
motors	number	Temperature on the direct output bus.
b5V	number	Temperature on the 5V bus.
b15V	number	Temperature on the 15V bus.
b24V	number	Temperature on the 24V bus.
b48V	number	Temperature on the 48V bus.
ambient	number	Ambient temperature.
lva	number	Temperature on the LVA power supply.
lvb	number	Temperature on the LVB power supply.
hva	number	Temperature on the HVA power supply.
hvb	number	Temperature on the HVB power supply.

For **transients** target:

JSON field	Type	Description
voltage	object	JSON description (see below) of the latest voltage transient.
current	object	JSON description (see below) of the latest current transient.

Transient object:

JSON field	Type	Description
battery	string	Latest transient on the input bus.
motors	number	Latest transient on the direct output bus.
b5V	number	Latest transient on the LVA bus.
b15V	number	Latest transient on the LVB bus.
b24V	number	Latest transient on the HVA bus.
b48V	number	Latest transient on the HVB bus.

For **supervisor_test** and **cta_test** targets:

JSON field	Type	Description
completed	boolean	True if the operation completed
start_time	number	Timestamp set at the beginning of the self-test.
end_time	number	Timestamp set at the end of the self-test.
progress	number	Progress of the self-test.
num_faults	number	Number of faults that have occurred until now.
faults	array	Array of 16 numbers describing the latest faults.

Setters:
None implemented for now

4.2.5.2. API backend

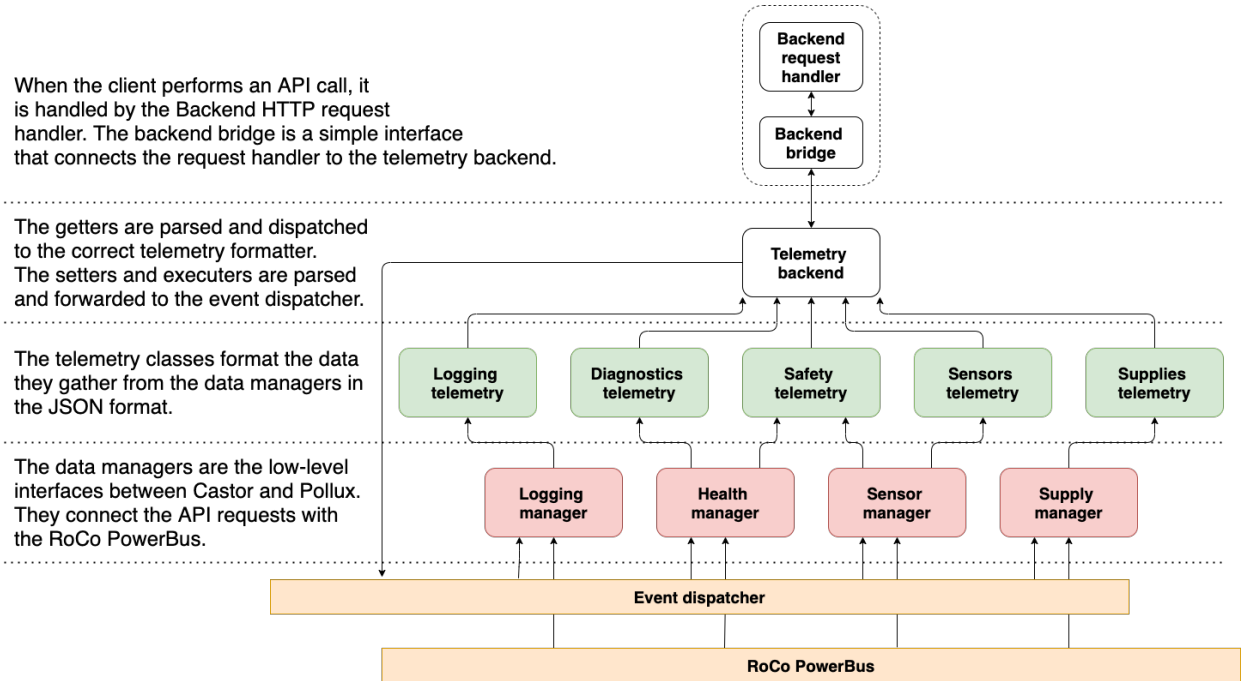
An HTTPRequestHandler in Backend.cpp/h handles client requests and parses the GET fields that encode the operation the client wants to perform. Three possible operations are available to the client: *exe*, *get* and *set*.
The backend bridge Backend.cpp/h is an interface that binds the backend request handler to the telemetry backend. For the sake of simplicity, only three functions are exported by the backend bridge:

```
std::function<bool(const char*, char*, uint32_t)> getter;
std::function<bool(const char*, const char*)> setter;
std::function<bool(const char*, const char*, char*, uint32_t)> executer;
```

These functions are bound to the following functions of the telemetry backend in TelemetryBackend.cpp/h:

```
bool get(const char* variable, char* result, uint32_t length);
bool set(const char* variable, const char* value);
bool execute(const char* action, const char* payload, char* result, uint32_t
length);
```

In these function prototypes, the fields `variable` and `action` correspond to the **target** and the fields `value` and `payload` correspond to the **value** in the API specification in section 4.2.5.1.
The telemetry backend dispatches the data requests (getters) to multiple telemetry formatters, such as DiagnosticsTelemetry.cpp/h, LoggingTelemetry.cpp/h, SafetyTelemetry.cpp/h, SensorsTelemetry.cpp/h and SuppliesTelemetry.cpp/h. The purpose of these files is simply to format the data gathered in the data managers in the JSON format, so that they can be used as a response to client requests.
The telemetry backend also dispatches the action requests (executers) to the EventDispatcher. Figure @ depicts the API backend architecture.



4.3. Additional software

4.3.1. API access

The easily access the Castor API, there is the possibility to add the following lines of code to your *.bashrc/.zshrc*:

```
xplore() {
    curl "http://192.168.1.42/api?a=$1&t=$2&v=$3"
}
```

After adding these lines to your *.bashrc/.zshrc*, do not forget to source it using:

```
source ~/.bashrc
```

or

```
source ~/.zsh
```

depending on whether the user uses a Linux distribution or macOS respectively (Windows does not count, please buy a decent computer).

These lines of code create a shortcut, so that instead of having to open a web browser and manually enter an API call for Castor, one can simply use the terminal.

For example, the following API call that resets the Pollux board:

```
http://192.168.1.42/api?a=exe&t=reset&v=cta
```

can be simplified by typing the following line in the terminal:

```
xplore exe reset cta
```

4.3.2. Power report generator

5. Operations

5.1. Tools and parts

Tools

Description
Electric tape [at least 1m]
Duct tape
Pen
AWG12 cable [at least 1m]
Weller Soldering iron
Weller SMD iron tip
Weller large cable iron tip
Lead-free solder
Flux with syringe
Weller tip activator
M1.5 Allen key
Multimeter
Adjustable 230VAC 5A 30V power supply
Oscilloscope with at least two channels sampling at no less than 100MSa/s
Swiss multi-socket
Epoxy tube

Parts

Reference	Description
BAT1	21V to 29V battery, including 30A BMS
BAT2	Same as above but with an incorrect discharge connector [!]
KEY	BMS charging activation connector
CHGR	Li-Ion battery charger
ACPS	1.5kW AC power supply 230V AC to 24V DC
ADP	European male Swiss female grid plug adapter
SAF	Safety circuit
PS3A	Pollux III power supply
PS3B	Pollux III power supply (backup)
SUP	Castor II supervisor WiFi module
LVA	Low-voltage (5V and 15V) power module
LVB	Same as above
HVA	High-voltage (24V and 48V) power module
HVB	Same as above
FAN	24V 120x60x18 double fan

DGL	Polarity inversion XT60 dongle
SCR	M1.5x8 screws [TO VERIFY]

5.2. Definitions

Generalities

Power stage. Part of a power module that which may nominally carry a high-current flow. Generally, it consists of large inductors, capacitors and MOSFETs.

Control stage. Part of a power module that controls the power stage by closing the loop between the produced voltage/current and the output MOSFET signals.

EMU (Energy Measurement Unit). Part of the power supply which computes how much power is dissipated by a given subsystem.

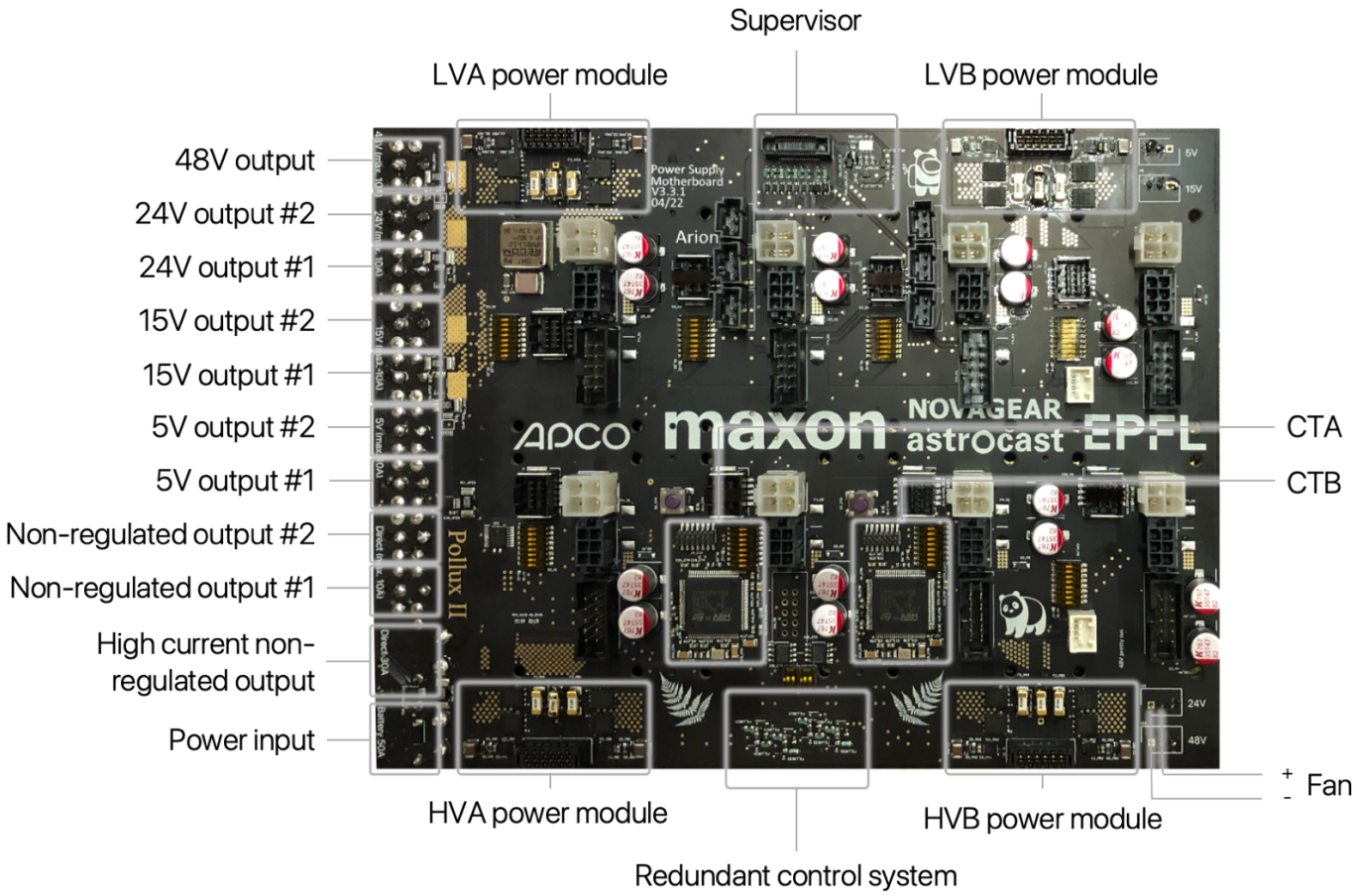
CTA/CTB (Controller A/B). Redundant micro-controllers that control the state of the power modules. In addition, CTA oversees the collection, logging, and transmission of the data from the EMU.

BMS (Battery Management System). Critical and intrinsic part of a battery pack. Ensures that the battery pack remains in its SOA (Safe Operating Area). Manages the charging and discharging process of the battery.

Ports

Reference	Description
BAT: DCH	Battery's XT60 male to discharge the power to the rover
BAT: CHG	Battery's XT60 female to charge the battery
SAF: DCH	Safety system's input from the battery
SAF: PSM	Safety system's output to the power supply
PS2: IN	Power supply's main input [30A]
PS2: DIR1	Power supply's direct output [30A] to the handling device
PS2: DIR2	Power supply's auxiliary direct output [10A]
PS2: 5V1	Power supply's 5V output 1 [10A] to the Science bay
PS2: 5V2	Power supply's 5V output 2 [10A]
PS2: 15V1	Power supply's 15V output 1 [10A] to the Main computer
PS2: 15V2	Power supply's 15V output 2 [10A]
PS2: 24V1	Power supply's 24V output 1 [10A] to the LiDAR
PS2: 24V2	Power supply's 24V output 2 [10A] to the Antenna
PS2: 48V	Power supply's 48V output [5A]
PS2: LVA	Power supply's LVA power connector
PS2: LVB	Power supply's LVB power connector
PS2: HVA	Power supply's HVA power connector

PS2: HVB	Power supply’s HVB power connector
PS2: FLW	Power supply’s front-left wheel connector group (CAN ID 1/5)
PS2: FRW	Power supply’s front-right wheel connector group (CAN ID 2/6)
PS2: BRW	Power supply’s back-right wheel connector group (CAN ID 3/7)
PS2: BLW	Power supply’s back-left wheel connector group (CAN ID 4/8)
PS2: SUP	Power supply’s supervisor connector
PS2: JTGA	Power supply’s JTAG connector for CTA
PS2: JTGB	Power supply’s JTAG connector for CTB
PS2: RSTA	Power supply’s reset button for CTA
PS2: RSTB	Power supply’s reset button for CTB
PS2: FLHA	Power supply’s flash button for CTA
PS2: FLHB	Power supply’s flash button for CTB
PS2: FAN+	Power supply’s fan positive pin
PS2 : FAN-	Power supply’s fan negative pin
LVA: CN	LVA’s power connector
LVB: CN	LVB’s power connector
HVA: CN	HVA’s power connector
HVB: CN	HVB’s power connector
SUP: CN	Supervisor’s connector to the power supply
SUP: USB	Supervisor’s micro-USB port to a host computer
SUP: RST	Supervisor’s reset button
SUP: FLH	Supervisor’s flash button
FAN: +	Fan’s positive terminal
FAN: -	Fan’s negative terminal



5.3. Operations procedure

5.3.1. Main procedure

Preconditions

a) Press on the emergency button.
b) Set the circuit breaker on ‘0’.
c) Remove the LVA, LVB, HVA, HVB, SUP modules from the power supply.
d) Disconnect the power supply’s polarity dongle from the safety circuit (don’t disconnect the dongle from the power supply).
e) Disconnect the battery “discharge” connector from the safety circuit.
f) Put electric type on the battery’s “discharge” male connector.
g) Put electric type on the battery’s “charge” male connector.

Preparation procedure [T - 1day]

a) Perform a battery integrity check with BAT1 [procedure: z)].
b) Charge the battery BAT1 [procedure: 5.3.3].
c) If any of [a b] fails, redo [a b] replacing BAT1 with BAT2.
d) Perform a safety system integrity check with SAF [procedure: -].
e) Perform a safety system quick check with SAF [procedure: 5.3.5].
f) If any of [d e] fails, proceed to the emergency procedure [procedure: TBD].
g) Perform a power supply integrity check with PS2A [procedure: 5.3.6].
h) Perform a power supply quick check with PS2A [procedure:5.3.9].
i) If any of [g h] fails, redo [g h] replacing PS2A with PS2B and skip [i j k l m u v w x].
j) Install the supervisor [procedure: 5.3.10].

k) Perform a supervisor SUP quick test [procedure: 5.3.12].
l) Perform a CTA quick test [procedure: 5.3.13].
m) Perform a CTB quick test [procedure: 5.3.13].
n) If any of [j k] fails, skip [u v w x].
o) Install LVA and LVB power modules [procedure: 5.3.14].
p) Perform a power module quick test with LVA/LVB [procedure: 5.3.15].
q) Install HVA and HVB power modules [procedure: 5.3.14].
r) Perform a power module quick test with HVA/HVB [procedure: 5.3.15].
s) If any of [p q] fails, abort the main procedure [procedure: 3.2] and proceed to the emergency procedure [procedure: TBD].
t) Connect subsystems [procedure: 5.3.16].

Mission procedure [T – 10min]

u) Start the power system [procedure: 5.3.7].
v) Monitor the power supply [procedure: 5.3.17].

Post-mission procedure [T + 40min]

w) Fetch the mission data [procedure: 5.3.18].
x) Generate mission report [procedure: 5.3.19].
y) Hand-out mission report file “report.pdf” in the generated Mission folder from step [w] to at least two ERC judges.
z) Stop the power system [procedure: 5.3.8].

5.3.2. Battery integrity check

Preconditions

a) Battery charge port is not connected.
b) Battery discharge port is not connected.
c) Multi-meter with leads is available.

Procedure

a) Start multi-meter.
b) Set multi-meter mode on DC (if applicable).
c) Connect multi-meter leads on V and COM ports.
WARNING: Failure mode [**Error! Reference source not found.**]: If the operator inadvertently connects any of the leads on the A/Ω port, a short-circuit will occur, the multi-meter’s fuse will be blown, the BMS will be bricked and the batteries could be damaged.
d) Set multi-meter on Voltage range 230V.
e) Measure the battery’s voltage on the BMS’ charge port with the multi-meter leads.
f) The battery integrity check succeeds if the voltage read on the multi-meter lies between 21V and 29V.

Failure modes:

- [**Error! Reference source not found.**]: BMS short-circuited
- [**Error! Reference source not found.**]: Undervoltage detected
- [**Error! Reference source not found.**]: Apparent overvoltage

5.3.3. Charging the battery

Preconditions

a) Battery charge port is not connected.
b) Battery discharge port is not connected.
c) Electric tape is placed on the discharge port.
d) Part CHGR (battery charger) available.
e) Part ACPS (230VAC to 24VDC power supply) available.
f) Part KEY (connector to enable BMS charging mode) available.

Procedure

a) Connect the KEY to the BMS. Ensure it is well secured into the BMS.
b) Connect the ACPS supply's XT90 (biggest connector on the ACPS) to the battery charger CHGR on port labelled DC-IN
c) Connect the BMS' charge port to the CHGR on port labelled CH1 or CH2
Three control buttons (TOP, MID, BOT) are available on the left or right for CH1 or CH2 respectively.
d) Press at least 0.5s on the MID button.
e) Using TOP and BOT, navigate to the "charge" menu.
f) Verify that the charger settings read as follows: {LiPo 4.2V, 7S, 10A, Charge}.
g) Select "Start" using TOP and BOT and validate using MID.
h) Press MID when the "Perform unbalanced task?" message appears.

Failure modes

- [**Error! Reference source not found.**] BMS full failure
- [**Error! Reference source not found.**] BMS detected overheat
- [**Error! Reference source not found.**] BMS detected overcurrent after short-circuit
- [**Error! Reference source not found.**] BMS detected overcurrent while nominally driving the rover
- [**Error! Reference source not found.**] BMS detected battery undervoltage
- [**Error! Reference source not found.**] BMS detected charger overvoltage
- [**Error! Reference source not found.**] Unable to charge battery
- [**Error! Reference source not found.**] Charger full failure
- [**Error! Reference source not found.**] Abnormal battery connection

5.3.4. Checking safety system integrity

Preconditions

a) Safety system's input is not connected.
b) Safety system's output is not connected.
c) Multi-meter with leads is available.

Procedure

a) Start multi-meter.
b) Connect multi-meter leads on Ω and COM ports.
c) Set multi-meter on Resistance range 1kΩ.

d) Measure the resistance on the safety system's input connector with the multi-meter leads.
e) If the resistance is less than 1kΩ, the safety system did NOT PASS the integrity check and the procedure terminates here.
f) Measure the resistance on the safety system's output connector with the multi-meter leads.
g) If the resistance is less than 1kΩ, the safety system did NOT PASS the integrity check and the procedure terminates here.

Failure modes:
- [**Error! Reference source not found.**]: Relay failure
- [**Error! Reference source not found.**]: Wiring failure

5.3.5. Safety system quick test

Preconditions
a) Safety system's input is connected to the BMS' discharge connector.
b) Safety system's output is not connected.
c) Multi-meter with leads is available.

Procedure
a) Start multi-meter.
b) Set multi-meter mode on DC (if applicable).
c) Connect multi-meter leads on V and COM ports.
WARNING: Failure mode [**Error! Reference source not found.**]: If the operator inadvertently connects any of the leads on the A/Ω port, a short-circuit will occur, even though there is a fast-acting circuit breaker, the multi-meter's fuse might still be blown. The BMS and batteries should remain protected, but the failure mode is still listed here for reference.
d) Set multi-meter on Voltage range 230V.
e) Press on the emergency button.
f) Set the circuit breaker on '0'.
g) Measure the safety system's output voltage with the two multi-meter leads.
h) The safety system quick test fails if the multi-meter indicates a voltage above 1V for more than 10s.
i) Set the circuit breaker on '1'.
j) Turn the emergency button to release it.
k) Measure the safety system's output voltage with the two multi-meter leads.
l) The safety system quick test fails if the multi-meter indicates a voltage less than 21V for more than 10s.

5.3.6. Checking power supply integrity

Preconditions
a) The power supply's input and outputs are not connected.
b) The power supply is not connected to the power modules LVA, LVB, HVA, HVB.
c) The power supply is not connected to the supervisor SUP.

d) Multi-meter with leads is available.

Procedure

h) Start multi-meter
i) Connect multi-meter leads on Ω and COM ports.
j) Set multi-meter on Resistance range 1kΩ.
k) Measure the resistance on the power supply's input with the multi-meter leads.
l) Measure the resistance on the power supply's 5V output with the multi-meter leads.
m) Measure the resistance on the power supply's 15V output with the multi-meter leads.
n) Measure the resistance on the power supply's 24V output with the multi-meter leads.
o) Measure the resistance on the power supply's 48V output with the multi-meter leads.
p) If the resistance is less than 1kΩ for any of [k l m n o] measurements, the power supply did NOT PASS the integrity check.

5.3.7. Starting power system

Preconditions

a) Safety system's input is connected to the BMS' discharge connector.
b) Safety system's output is connected to the power supply's input connector.

Procedure

a) Set the circuit breaker on '1'.
b) Turn the emergency button to release it.

5.3.8. Stopping power system

Procedure

a) Press on the emergency button.
b) Set the circuit breaker on '0'.

5.3.9. Power supply quick test

Preconditions

a) The power supply's outputs are not connected.
b) The power supply is not connected to the power modules LVA, LVB, HVA, HVB.
c) The power supply is not connected to the supervisor SUP.

Procedure

a) Start power supply according to [procedure: 5.3.7].
b) The quick test succeeds if a bright LED next to the SUP connector can be observed.
c) Stop power supply according to [procedure: 5.3.8].

5.3.10. Installing supervisor

Procedure

a) Insert the supervisor SUP on its socket connector PS:SUP.
   By design, it is not possible match the connectors wrongly.
   It is possible to have the impression that the connector is not well-secured into its socket. This is only an impression.

5.3.11. Connecting to supervisor

Preconditions

a) Supervisor was installed according to [procedure: 5.3.10].
b) Power supply was started according to [procedure: 5.3.7].
c) A computer/tablet/smartphone (DEV) is available.

Procedure

a) Ensure that the LED on the supervisor is blinking (at around 1Hz).
b) Connect to WiFi or 4G with your DEV.
c) Open the Slack application.
d) Browse to the EPFL Xplore slack workspace.
e) Send a direct message to yourself with content “get_castor_password”.
f) Answer the security question and note the obtained password.
g) Using a computer, tablet or phone, connect to the WiFi network called “Xplore_2.4GHz”, using the password obtained in (d).
h) Ensure that the LED on the supervisor is now constantly green.
i) Open a web browser and follow the link http://192.168.1.42/
j) If a user interface appears, the operator was successfully connected to the supervisor.

5.3.12. Supervisor quick test

Preconditions

a) Operator is connected to supervisor according to [procedure: 5.3.11]

Procedure

a) The quick test succeeds if the “Supervisor: Ping” in the “Diagnostics” panel has a value between 10ms and 500ms.

5.3.13. CTA/CTB quick test

Preconditions

a) Operator is connected to supervisor according to [procedure: 5.3.11]

Procedure

a) The quick test succeeds if the “CTA/CTB: Ping” in the “Diagnostics” panel has a value between 0.1ms and 10ms.

5.3.14. Installing power module

Procedure

a) Insert the given power module on its socket connector.

LV power modules can be recognized by the Silver colour of the vias and pads.

HV power modules can be recognized by the Gold colour of the vias and pads.

LV power modules are matched to PS:LV connectors (see definitions [5.2]), on the side of the SUP WiFi supervisor.

HV power modules are matched to PS:HV connectors (see definitions [5.2]), on the side of the CTA/CTB microcontrollers.

5.3.15. Power modules quick test

Preconditions

a) The power supply's outputs are disconnected.
b) The power supply is connected to the supervisor SUP.

Procedure

a) Start power supply according to [procedure: 5.3.7].
b) Connect to the supervisor according to [procedure: 5.3.11]
c) Ensure that the output voltages of the power module being tested are displayed on the "Supplies" panel in the user interface.
d) If the voltages are withing 5% of the target voltage and a green label "Stable" is displayed for each of the two voltages, then the quick test succeeded.
e) Stop power supply according to [procedure: 5.3.8].

5.3.16. Connecting subsystems

Preconditions

a) There are no short-circuits on the subsystems.

Procedure

a) Connect the subsystem's DC jacks in the power supply, according to the subsystem's voltage needs.
b) Use zip-ties between the DC jacks to create a small lateral constraint on the connectors.

5.3.17. Monitoring the power supply

Preconditions

a) The power supply's outputs are connected to the subsystems.
b) The power supply is connected to the supervisor SUP.

Procedure

a) Connect to the supervisor according to [procedure: 5.3.11]
b) Ensure that the operator is successfully connected to the user interface.

The battery charge and remaining operational time are display on the "Battery" panel.

The output voltages of the power module being tested are displayed on the "Supplies" panel.

The peak temperatures and currents are displayed in the "Safety panel".

c) Inform the rest of the operational team when the battery charge reaches the following values: [50% 20% 10% 5% 2% 1%], along with the estimated remaining operational time.
d) If any voltage is not within 5% of the target voltage or a red label “Unstable” is displayed on any voltage, failure mode [] is reached.
e) If the rover’s peak temperature is higher than 70°C, failure mode [] is reached.
f) If the rover’s peak current is higher than 30A, failure mode [] is reached.
g) If “Supervisor: Ping” is over 500ms, reset the supervisor in the “Diagnostics” tab.
h) If “CTA: Ping” is over 50ms, reset CTA in the “Diagnostics” tab.
i) If “CTB: Ping” is over 50ms, reset CTB in the “Diagnostics” tab.
j) If “Supervisor: Ping” remains over 500ms after resetting, disconnect and reconnect to the Xplore_2.4GHz network.
k) Repeat [c d e f g h i j] until the end of the mission.

5.3.18. Fetching mission data

Preconditions

a) The power supply is connected to the supervisor SUP.

Procedure

[TODO] Implement a way to know which mission ID we are in.
a) Download the file http://192.168.1.42/download?t=cta&m=0
b) Download the file http://192.168.1.42/download?t=cta&m=1
c) Download the file http://192.168.1.42/download?t=cta&m=2
d) Download the file http://192.168.1.42/download?t=cta&m=3
e) Download the file http://192.168.1.42/download?t=cta&m=4
f) Download the file http://192.168.1.42/download?t=cta&m=5
g) Download the file http://192.168.1.42/download?t=cta&m=6
h) Download the file http://192.168.1.42/download?t=cta&m=7
i) Download the file http://192.168.1.42/download?t=cta&m=8
j) Download the file http://192.168.1.42/download?t=cta&m=9

5.3.19. Generating mission report

Preconditions

a) The power supply is connected to the supervisor SUP.
b) XploreGrapher is installed in the device DEV.

Procedure

a) Retrieve the mission ID (TODO) and denote it by [mission_id].
b) Run the following command: “./XploreGrapher.py cta [mission_id]”.
c) The mission report PDF document, along with the LaTeX source files, are now located in the “Mission [mission_id] (cta)” folder.

5.4. Hardware integration procedure

5.4.1. Main procedure

5.4.2. Integrating battery

5.4.3. Integrating safety systems

5.4.4. Integrating power supply

5.5. Software integration procedure

5.5.1. Main procedure

5.5.2. Updating BMS firmware

5.5.3. Integrating supervisor software

5.5.4. Integrating CTA/CTB software

5.5.5. Installing XploreGrapher software