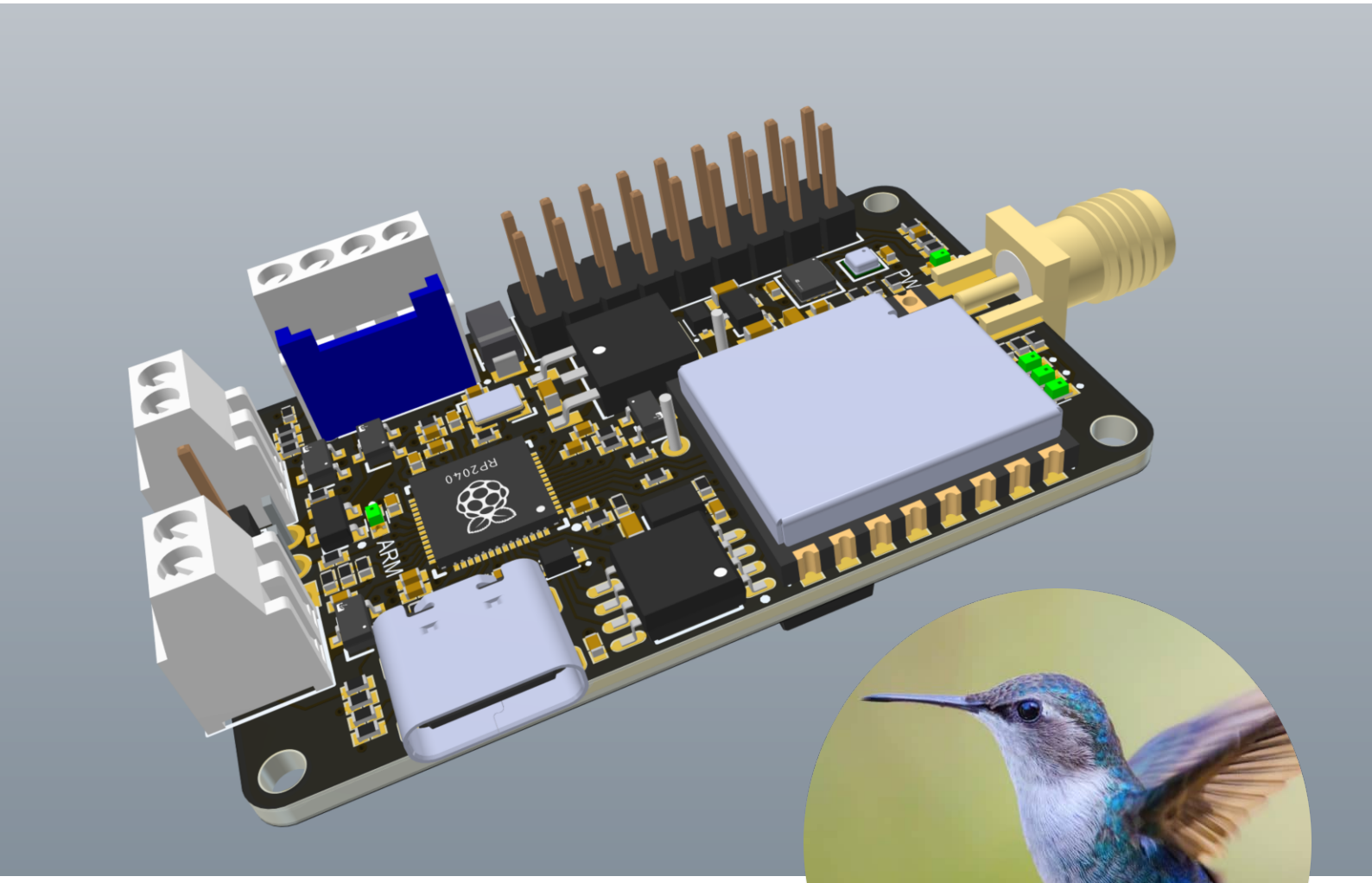
DARE
Delft Aerospace Rocket Engineering

The Kolibri Flight Computer

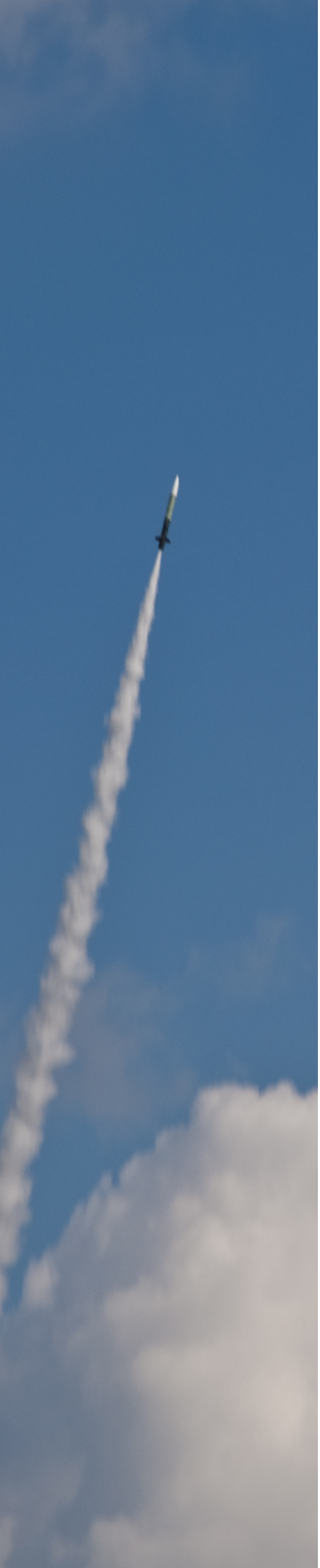
Design documentation and development notes for a compact and inexpensive flight control board



Authors:
EEEEEEEEEEEE

Version:
V 0.1.0

Friday 16th May, 2025



DARE
Delft Aerospace Rocket Engineering

Contents

1 Introduction

Kolibri (*noun*): Word for *hummingbird* in a several European languages

The flight computers within the Delft Aerospace Rocket Engineering (DARE) are roughly of two types. One is the SRP board, a relative small and inexpensive 5x5cm hexagonal PCB with timer-based deployment, and larger, often multi-stack systems for larger rockets such as the Cansat or Stratos. For smaller experimental rockets, the SRP board is often the option of choice, but the lack of onboard sensors and data logging makes post-flight or post-failure analysis difficult. Furthermore, the implementation of data telemetry within smaller DARE projects is rare, at times leading to rockets disappearing without any data.

The goal of this project is to develop a flight control board optimized for size and cost. In particular, the design goals are:

- If possible, size less than 50x30 mm
- Can work in place of the SRP board
- Base cost of max 20 EUR per board in batches of 5 (no telemetry)
- Can be ordered mostly pre-assembled
- Idiot-resistant (tm) with safety features against shorts, reverse voltage, or software bugs.
- Barometer
- Inertial Measurement Unit
- On-board data logging
- Optional support for long range telemetry, with the board doubling as a ground station.
- Feature comparable with the Altus Metrum TeleMini (150 USD).

Stretch goals (only if they can be done for cheap)

- GPS support
- Dual deploy/dual servo

During the design process, the following are key factors guiding the design choices:

- Cost
- Size
- Component availability (Jellybean parts with drop-in alternatives are preferred)
- Reliability (It should be resistant to common failure modes)
- Diagnostic flight data regardless of outcome/failure mode
- Ease of use
- Flexibility
- Laziness

The following are *not* part of the core goals:

- Ease of manual assembly
- Ease of repair/debugging

Due to the compact target size, space for component designators will be limited. Repairs will likely require some soldering experience due to the density. If all goes well, the result will be more akin to a "commercial" medium density board like the Arduino Nano RP2040 Connect, which is not meant to be repaired, and only requires the user to solder optional pins.

If successful, this can serve either as a primary flight computer for space-constrained rocket designs, or as an add-on to augment an existing flight computer with data logging and telemetry. A similar flight control board was previously developed by the author (STM32, Baro, IMU, Servos, LoRa transceiver, SD/flash blackbox), which was successfully used in SRP, but had a significantly higher unit cost and required manual assembly due to some uncommon RF

components. This project aims to achieve a similar functionality at the lowest possible cost and minimal labor required for assembly. The target size is also literally the approximately length of a Bee hummingbird[1] so ehhhh, the project name was totally on purpose :)

This document serves as documentation of the design process, which may be of use to others wanting to design something similar. The first part of the document will lay out the top level design choices, including I/O, sensors, actuators, and requirements, while the rest of the document contain notes during the component selection and PCB design process. Do note that this document may be quite a mess, since it also serves as a notebook for random notes. If there are random numbers somewhere, just ignore them. If you do have any questions, feel free to reach out to the author.

2 Requirements and IO

For starters, the features of the SRP board:

- Reverse voltage protection
- Up to 10 volt (1s or 2s lipo)
- 1x Servo connector
- 1x Pyro actuation channel
- 1x External LED
- Breakwire (external arm)
- Physical arming switch terminal
- Arming measurement
- Continuity measurement
- UART interface
- terminal connections: bat(2x), Arm(2x), Servo1/pyro1(3x),Servo2/pyro2(3x, only 2x for pyro terminal)

These will be the guiding requirements for now. Adding to this, the other requirements are:

- Onboard baro and IMU sensors
- Onboard data logging
- Support for optional telemetry
- Support for optional GPS

Based on these, the new requirements for the project will be:

- Reverse voltage protection
- Up to 8.4 volts (2s lipo)
- at least 1x servo connector
- at least 1x pyro actuation channel with continuity measurement
- UART interface
- USB interface
- physical arming
- barometer sensor
- imu sensor
- onboard data logging
- Optional telemetry
- optional GPS
- Indicator LED's
- Buzzer

These are the main hardware requirements. Other goals, e.g. board size and cost, are sort of flexible, but they should be "as small/low as possible"

[1] https://en.wikipedia.org/wiki/Bee_hummingbird

3 Cost-saving strategies

For PCB manufacturing and assembly, one of the cheapest available options is JLCPCB based in China. They offer an assembly service using parts from their extensive component library[2]. As a cost-saving measure, an attempt is made to source as many parts from this library as possible. The JLCPCB library contain both *basic* and *extended* items, the later of which cover less common components and incurs an additional 3 USD charge per new item type. Whenever possible, items in the *basic* category are selected. If possible, a four layer PCB under 50x50mm are basically free.

When selecting between similar parts, the one with the most units in stock is usually taken, hopefully decreasing the chance of a sudden lack of availability.

JLCPCB offers economical and standard assembly options. The economy option (8 USD) does not allow tiny footprints and is limited to one sided assembly, while the standard option is about 25 EUR. Parts are thus selected to conform with the economy assembly guidelines.

Acquiring parts from different vendors requires additional shipping fees. Therefore, some parts are selected based on minimising the number of orders from different sources.

4 Component selection

This section contains notes about the selection of the main components.

4.1 MCU

The requirements for the microcontroller is pretty basic. It needs enough GPIO pins for the actuation and sensing, and it needs communication interfaces. The only *basic* microcontroller options from JLCPCB are some STM32 microcontrollers, but due to the use of a RP2040 development board in the SRP2020 PCB, the same RP2040 microcontroller is selected for this design. This option is cheap (1.1 EUR), has existing developed software, and is compact.

The RP2040 is a dual core arm-based controller from the Raspberry pi foundation. Although it's an M0+ core without an FPU, no intensive floating point heavy math is expected. Other microcontrollers (STM32, esp32 etc) are available at the same price point, but this strikes a nice balance between performance, cost, and features. It also has some neat features such as the PIO state machines, thus allowing the emulation of peripherals or protocols on arbitrary GPio pins.

4.2 Flash/blackbox

The RP2040 required an external flash chip for program memory. If there's enough space, this chip can also serve as non-volatile storage space to log flight data.

The only QSPI flash chip available as a *basic* part for JLCPCB assembly[3] is the W25Q128JVSIQ (0.76 EUR), which runs at the right clock speed and is coincidentally the same part used by the RP2040 hardware design manual[4]. So this is the obvious choice.

Flight data recording should be able to handle multiple flights.

4.3 Sensors

For sensors, it is decided to use SPI as the primary communication protocol due to its speed compared to I2C. This reduces the wait time and delay when acquiring the values. The main

[2] https://jlcpcb.com/parts
[3] https://jlcpcb.com/parts/1st/Memory_34
[4] https://datasheets.raspberrypi.com/rp2040/hardware-design-with-rp2040.pdf page 8

Name	Manufacturer	Interface	Min pressure (hPa)	Cost (USD)
LPS22HBTR	STmicroelectronics	I2C/SPI	260	1.31
HP203B	HopeRF	I2C	300	1.01
MPL3115A2R1	NXP	I2C	200	5.32
BMP388	Bosch Sensortec	I2C/SPI	300	2.55
BMP280	Bosch Sensortec	I2C	300	1.76
SPL06	Goertek	I2C	300	1.21

Table 1: Baros

Name	Manufacturer	Interface	Cost (USD)
LSM6DS3TR-C	STmicroelectronics	I2C/SPI	1.32
LSM6DSRTR	STmicroelectronics	I2C/SPI	1.95
MPU-6050	Invensense	I2C	11.9
QMI8658C	QST	I2C/SPI	2.55
MPU6500	Invensense	SPI	3.05
QMI8658A	QST	I2C/SPI	1.57
LSM6DSOWTR	STmicroelectronics	I2C/SPI	1.34
BMI160	Bosch Sensortec	I2C/SPI	5.4
BMI055	Bosch Sensortec	I2C/SPI	1.60
BMI088	Bosch Sensortec	I2C/SPI	3.28
BMI270	Bosch Sensortec	I2C/SPI	1.83

Table 2: imus

sensor of interest during flights is the barometer, which gives an indication of the altitude during the flight. Unfortunately no barometers are available as *basic parts*. Select pressure sensors available at the time of writing are shown in Table 2.

The author had previous experience with SPL06, but since the LPS22 from ST is available with SPI and is inexpensive, this one is initially selected. This may later be replaced with the BMP388 if the costs allow, which is overall a better sensor with the same overall dimensions. Note that the minimum pressure for all options correspond to approximately 10 km.

Update: The pressure sensor has been replaced with the BMP388. It's only slightly more expensive and offers an incredible performance compared to the LPS22. E.g. it has a relative pressure accuarcy of 8 Pa and an absolute pressure accuracy of 50 Pa which is pretty incredible. This is useful for getting an accurate altitude in dynamic environments. The pinout and footprint of the BMP388 is also identical to that of the LPS22, so they are drop-in replacements of each other.

For the inertial measurement unit, the main options are: For a low cost option, the IMU's from ST offer a nice balance between performance and cost in a compact package. The ones available are limited to +/- 16 G, but the pin compatible lsm6dso32 offers up to +/- 32 G of acceleration measurement.

From speaking with drone and rocketry hardware developers, the Bosch sensors offer improved noise immunity when compared to the sensors from ST. The BMI088 in particular is developed specifically for harsh environments with heavy vibrations found in drones and robotics (and rockets). Furthermore, of the available options, it offers the highest acceleration range at +/- 24 G[5]. For almost any other project, the BMI088 would be the ideal choice, but since the ST sensors are more compact, cheaper, and offer adequate signal quality for basic sensor logging, one of the LSM6DS* sensors are used. If you are designing a board, and the BMI088 is available, just pick that one. It's pretty great and performs way above its class. If the space and budget allows, the ST IMU may be replaced with the BMI088 in this design as well.

Update: Turns out the BMI270 is also available for only slightly more than the LSM6 chip. The

[5] https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi088/

BMI270 has a similar performance to the BMI088[6] but does not feature the advanced vibration filtering of the BMI088. The BMI270 is being used in some modern drone flight controllers[7] and has some features to allow for decent noise supression. Compared to the BMI088, the BMI270 is also a bit more compact and requires routing less signals (gyro and accel are combined as opposed to having separate chip select pins). It also uses the same pin mapping/footprint as the LSM6 chips so it's still easily swappable. So let's swap to using the BMI270 and the IMU.

4.4 Telemetry transceiver

With the optional telemetry transmitter, it should be compact, light, and have as high of a transmit power as possible and allowed. Of all the european license-free frequency bands, a narrow part of the 868 MHz band allows for 500 mW of transmit power. 2.4 GHz is limited to 100 mW, while most other frequencies are limited to 25 mW. Since lower frequencies experience less attenuation, the 868 MHz band provides the longest theoretical range. Examples of transceivers capable of operating at these frequencies are the CC1200 (40 mW, used on altus metrum products, FSK), Semtech SX1272 (100 mW, LoRa and FSK), SX1262 (150 mW, LoRa and FSK). For flexibility and range, one of the Semtech transceivers are ideal due to the high transmit power and support for LoRa modulation (ultra long range spread spectrum). Semtech transceivers often found in amateur high altitude balloons and cubesats. Some previous projects within DARE have also used RFM95 LoRa modules at the same frequency band, so perhaps there are already some antennas around?

With the sx1262, both integrated transceiver modules, and the IC itself are available. The modules are about 5 EUR, while the chip is only 3 EUR, but requires external front end circuitry. After some trial and error, it turns out that almost all of the component values required for the front end are only available as *extended* components in the JLCPCB library, leading to a final cost of about 10 EUR per transceiver (in batches of 5), even with significant BOM optimization. The downside to using integrated modules is a larger footprint, but as a cost saving measure, the transceiver module is used. This also makes it easy to deselect the transceiver during the assembly ordering process if a board without RF functionality is desired.

There are SMT SX1262 modules available with (almost) the same footprint as the common RFM95 modules, so if the module is no longer available, others serve as (almost) drop-in replacements supporting the same frequency and protocol (might need to change some wires)[8] Examples of drop-in replacements are with their transceiver IC's and transmit powers are:

- Ai-Thinker RA-01H (SX1276, 100mW)
- Ai-Thinker Ra-01SH (SX1262, 150mW)
- DreamLNK DL-RFM95-868M (SX1276, 100mW)
- HopeRF RFM95 (RF95, 100mW)
- G-NiceRF LORA1262 (SX1262, 150mW)
- Vollgo SX1262S8S+T-X1 (SX1262, 150mW)

Though of course double check the pinouts before replacing the chip. Even though they all have the samme footprint, some have the pins swapped around...

Out of these, the Vollgo module, the Ra-01SH, and the DL-RFM95-868M are available for assembly. Initially the SX1262S8S+T-X1 was used in the schematic/PCB design, but it has an impressively awful datasheet with ambiguous pin mappings. Seems like they took the datasheet from a previous design and forgot to change some text... (e.g. it specifies that the SPI data out pin is an input, and one pin is mapped to the BUSY pin, or the DIO pin, depending on what section you're reading...) Later it turned out that the Ra-01SH module is slightly cheaper (4.88

[6] https://community.bosch-sensortec.com/t5/MEMS-sensors-forum/BMI088-vs-BMI270-which-has-the-best-long-term-stabil
m-p/58769?attachment-id=4248

[7] E.g. in-depth dive here: https://www.youtube.com/watch?v=UWiysMidBHM

[8] See here for SMT options: https://jlcpcb.com/parts/2nd/iot_communication_Modules/Lora_Modules_
80470

EUR instead of 5.34 EUR), has more units in stock, and uses the same chip, so let's use that one.

Antenna examples: https://medium.com/home-wireless/testing-lora-antennas-at-915mhz-6d6b41ac8f1d

Maybe take some inspiration from the Altus Metrum AltOS telemetry packets? https://altusmetrum.org/AltOS/doc/telemetry.html#_history_and_motivation.

In ground station mode, just listen to packets and spit the raw bytes as hex over serial. That should work fine.

4.5 GPS

The L80RE is the cheapest choice. It's used on some eggtimer products, but the performance is not great in dynamic scenarios. It'll give a rough flight path, but probably very noisy. The main advantage of adding GPS is as an aid in finding the landing location of the rocket. Some links about GPS performance on rockets:

https://www.rocketryforum.com/threads/status-of-eggtimer-rocketry-gps-products.171742/ https://discord.com/channels/723644976638066845/723662607831007293/923401690756038676 https://www.rocketryforum.com/threads/quectel-l80.158692/ http://jcrocket.com/gps-tracking.shtml

Since communication occurs over UART using NMEA strings, a better GPS module can be hooked up to it if dynamic tracking performance is required. Since the ground station uses the same hardware, it would be possible to calculate the distance between the ground station and the rocket as a way of helping find it.

This document details how U-blox GPS are fairly reliable for high dynamic rocket flighst: http://tripoli-records.org/records/u-blox-GPS.pdf

Video showing how expensive GPS receivers are better: https://www.youtube.com/watch?v=QfwV7NP5BI8

Unfortunately during the design process, the L80RE GPS module went out of stock. This is fine, since the GPS is expected to be a separate board from the main flight computer (to allow for optimal antenna positioning), but unfortunately it now cannot be ordered with the other stuff :(. GPS modules have pretty standard interfaces, typically ground, 3.3V, TX, and RX. So if no module is designed, ready-made modules can be used as well. They are quite a bit more expensive though...

Other potential lost-cost options:

- https://www.lcsc.com/product-detail/Satellite-Positioning-Modules_Vollgo-VG7669T160N0MA_C5197313.html
- https://www.lcsc.com/product-detail/Satellite-Positioning-Modules_EXTREME-POWER-RC12_C5155829.html

This is not relevant for this project, but the MAX2769 universal GPS receiver allows for software-based baseband decoding. It's apparently also possible to decode it with an SDR and GPS antennas: https://www.rtl-sdr.com/updates-on-using-an-rtl-sdr-for-gps-on-a-high-powered-roc

4.6 Power supply

8.4v input, 3.3V output, size for 200 mA. Telemetry uses max 120 mA, GPS uses 25 mA. Sensors use a couple mA. The ams1117-3 ones would be ideal, but it's kinda bulky. Maybe move to bottom side.

Most standalone GPS modules use 5V (with an onboard 3.3v regulator). The board itself doesn't really need 3.3V, but but splitting the regulation up into a 5V regulator and 3.3V regulator, the maximum current capability is increased (each chip has a limited maximum heat dissipation).

Options for buck converters were looked into, but this would increase the complexity/cost of the system. In the space-constrained board layout, compact (i.e. high frequency) buck converters may also interfere with RF reception.

The AMS1117 requires using tantalum capacitors for stability, which is kinda annoying. Assuming a maximum power dissipation of 1.2W and an input power of 8.4V, the maximum current from an AMS1117-5 is 1.2/(8.4-5) = 0.35 A. If directly regulating to 3.3V, the max current is 1.2/(8.4-3.3)= 0.235 mA. The dropout voltage of the AMS1117 is about 1V, meaning it can't get 3.3V from a 1s lipo.

A SOT-23 package has a maximum power dissipation of 0.25 mW. Regulating from 5 to 3.3V gives 0.25/(5-3.3) = 0.147 A of continuous current. Since these are compact, one sot-23 regulator can be used to supply the RF chip, while another provides power to the rest of the system.

Another option is the L78M05 from ST-microelectronics, it's in a DPAK package, so slightly larger than the 1117, but can work fine with ceramic capacitors. It provide up to 500 mA of current and isn't power dissipation limited: 0.5 * (8.4-3.3) = 2.55 W, which is below the DPAK dissipation.

Using a 5V regulator also exposes a 5V connection, which might be useful for companion boards/modules.

The XC6206P332MR from Torex Semiconductor has a typical dropout of 250 mV (max 680 mV) at a load of 100 mA, meaning it can work with 1s lipos. It's also available as a basic smd part. Two of these can be added, one for the RF transceiver and one for the rest of the system.

5 Misc. non-SMT components

These have to be assembled by hand afterwards:

- Battery screw terminal
- Pyro terminals or servo header
- Extra indicator LED.
- Loud buzzer
- Voltage regulators?

For the USB connector, USB C is chosen for it's durability and abundance compared to Micro-USB.

For buttons, only boot for programming maybe? There's not really a need to include the reset button since that can be done through software. Apparently it's also possible to reboot into bootloader mode through the SDK, but eh, still nice to hav a physical button: https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_stdio_usb/include/pico/stdio_usb.h#L47

For the pyro connections, 2.54mm screw terminals are used.

Ordering from LCSC:

- Screw terminal: DIBO DB125-2.54-2P-GN-S
- Buzzer: KXG1205C
- LM317T: https://www.lcsc.com/product-detail/Linear-Voltage-Regulators-LDO_STMicroelectronics-LM317T-DG_C18718.html
- Generic 1117-3.3 LDO: https://www.lcsc.com/product-detail/Linear-Voltage-Regulators-LDO_UMW-Youtai-Semiconductor-Co-Ltd-AMS1117-3-3_C347222.html

Ordering from farnell:

- Screw terminal: 2112482
- LM317T: 9756027
- any 1117-3.3 LDO: e.g. 1652366, 1825376
- Buzzer, KXG1205C: 2215080

6 Pyro actuation circuit

The electronics subsystem of a model rocket may range in complexity from a simple timer to fancy setups with data logging and active control, although all of these need to somehow activate a recovery system. A popular method is by using a pyrotechnic charge, which has proven to be both a simple and reliable deployment method. Alternatively, mechanical actuators are another possible approach, but this won't be discussed in this document.

A pyrotechnic charge consists of a small amount of low explosive such as black powder, the combustion of which results in rapidly expanding gasses ejecting the recovery equipment. To activate such a pyro charge, the black powder must be ignited somehow, this could be using a fuse, although the focus will be on electronic methods. The firing of the pyro charge requires heat, which is typically created through a resistive load. Home built solutions may utilize nichrome wire, while commercial solutions etc.

To activate the ematches, a certain amount of current has to flow through the resistive load. When the temperature is sufficiently high, the charge is ignited.

Although the thermal power released can be calculated using this isn't enough to iej

6.1 Evaluation of existing designs

6.1.1 SRP board

Two versions of the SRP board are currently known. One was used before 2022 based around an attiny MCU (let's call that v1). Due to component shortages, a redesigned version based on the rp2040 was produced (let's call that v2). Both versions used almost the same pyro channel topology, with some component changes.

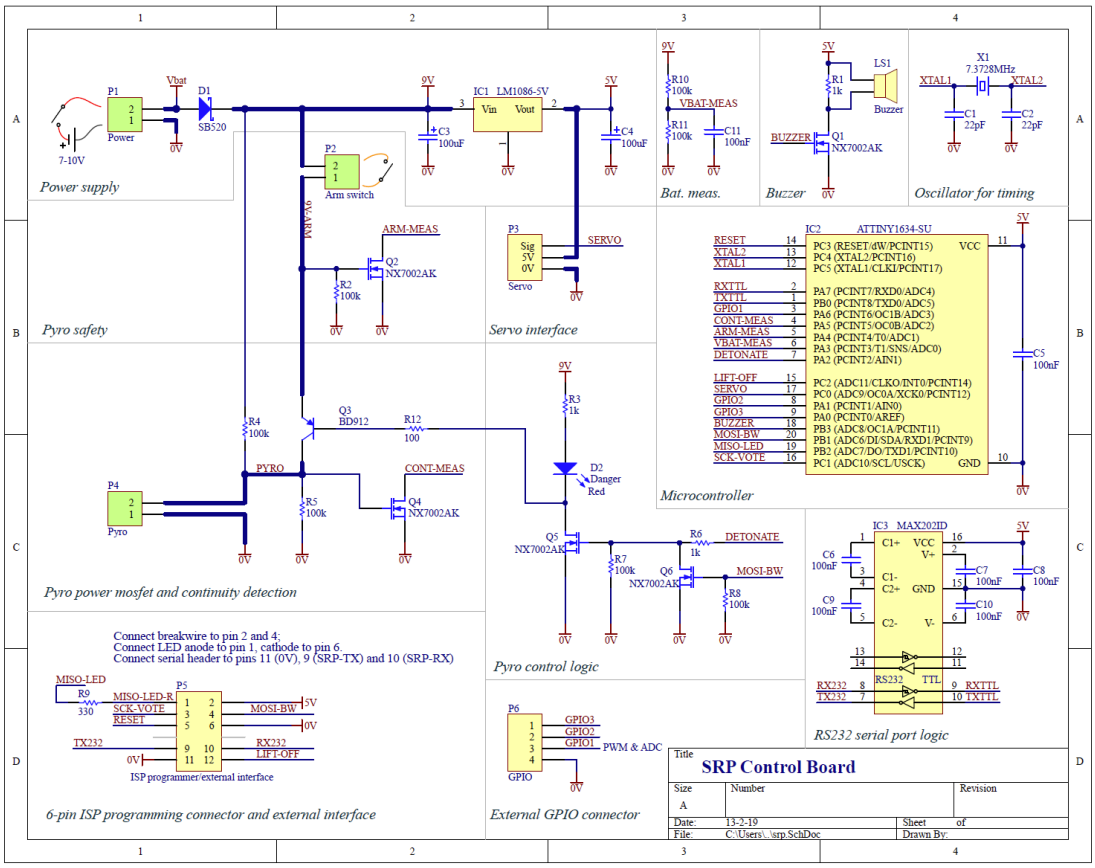


Figure 1: SRP board schematic

One of the most common failure modes of previous SRP boards[9] was a deliberate short of the pyro channel during testing to simulate the presense of a e-match. This was in fact part of the recommendation in the SRP electronics manual v3.5.0. The state machine is also designed such that once the pyro channel fires, it continues to be active[10]. Suppose the SRP board is powered by a fully charged and healthy 2s lipo at 8.4 volts, and suppose it can deliver 10 amps. Suppose furthermore that an SRP'er is testing everything and has shorted the pyro channel. Here's what happens once deployment occurs in the state machine:

1. Detonate pin signal is sent
2. Q5 activates, pulling current through R12 and the base of Q3, the main transistor.
3. The current through R12 is approximately (8.4 - 0.7)/100 = 0.077 A = 77 mA.
4. The BD912 PNP transistor amplifies the current roughly 40 times (based on the DC current gain graph), leading to 3.08 amps of current flowing through the system.
5. Since the pyro is treated as a short, the full voltage is present over the transistor.
6. 8.4 volts at 3 amps is 24 watts of heat, dissipated by the transistor.
7. TO-220 packages have a thermal resistance to air of approximately 60 60 °C W^{-1} (referenced from other part). Assuming ambient temperature of 25 degrees celsius, the transistor heats up to 1465 °C. In other words, it fails.
8. The transistor can either fail in open or short, if it fails in short circuit, the LiPo is being discharged at the full capacity, until a trace acts as a fuse.

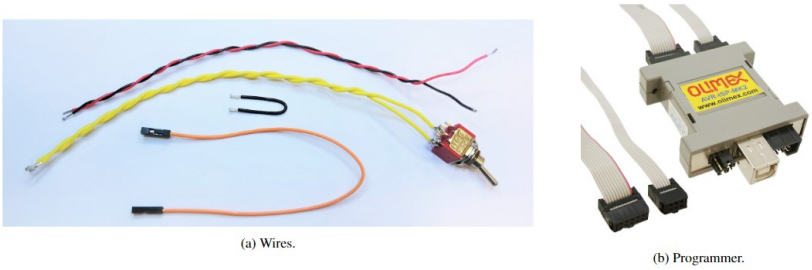
Not sure if documented at the time, but evidence point to this failure mode happening, in once

[9] author's observations during the SRP 2021 and second-hand gossip from SRP 2022

[10] https://git.projectstratos.nl/Electronics/srp/src/branch/master/Software/src/state_machine.cpp#L150

Hardware Setup Follow the below steps to setup the SRP PCB, the programmer and wiring correctly, so that the microcontroller can be flashed via Atmel Studio and tested right after. It strongly recommended to create a clean and spacious work area so you don't lose components or end up damaging your hardware by accident. In addition, wear an ESD wristband if possible and if you are not able to find some of the mentioned parts, ask for help!

1. Make the following wires (Fig. 9a):
   - a pair of red a black wires stripped on both ends to power your SRP board;
   - a short stripped wire to emulate a pyro for the SRP board; and
   - a pair of stripped wires to emulated the ARM switch.
2. Collect the following items:
   - a simple wire with connections on either side that can go on the breakwire connector;
   - an Olimex programmer (Fig. 9b) with its USB and programming cables; and
   - a power supply with clips to connect to your pair of power cables (or a battery that can also power SRP).
3. Make sure that the Olimex programmer is correctly configured: refer to the yellow sticker on the programmer and set the TARGET jumper in OFF mode.
4. Connect the programmer to your computer. Necessary drivers should automatically install.
5. Check the settings of the power supply: set it at 9V, and limit the current to 0.5 A (this current limit is a little bit conservative for normal operation, but for programming it will be definitely sufficient).
6. Turn off the power supply and connect your self-made power cables to the "Power" port of the SRP board and attach clips of the power supply to the other end of the wires.



(a) Wires.

(b) Programmer.

Figure 2: Testing procedure for v1 SRP board

case, the main power transistor had shorted junctions. In another case (v2 board), traces were burnt in the current path, indicating severe over current conditions. Note that the v2 board had a smaller SOT23 transistor which is expected to overheat and/or fail faster than the V1 board. Shown on figure Figure 2 are the testing procedures for the v1 SRP board. Due to the PSU current limit of 0.5 amps, nothing will break here, but if a test is conducted with flight batteries, chances are that things might break. Just speculating, but this could be the reason for last minute failures of the electronics when tested using flight hardware. In the manual for the v2 SRP board, a LED with resistor is suggested to emulate the pyro instead of a wire.

6.1.2 Stratos IV

The Stratos IV cutter actuation schematic[11] is shown in Figure 3:

[11] https://git.projectstratos.nl/Electronics/stratos-iv/src/branch/master/Hardware/REC

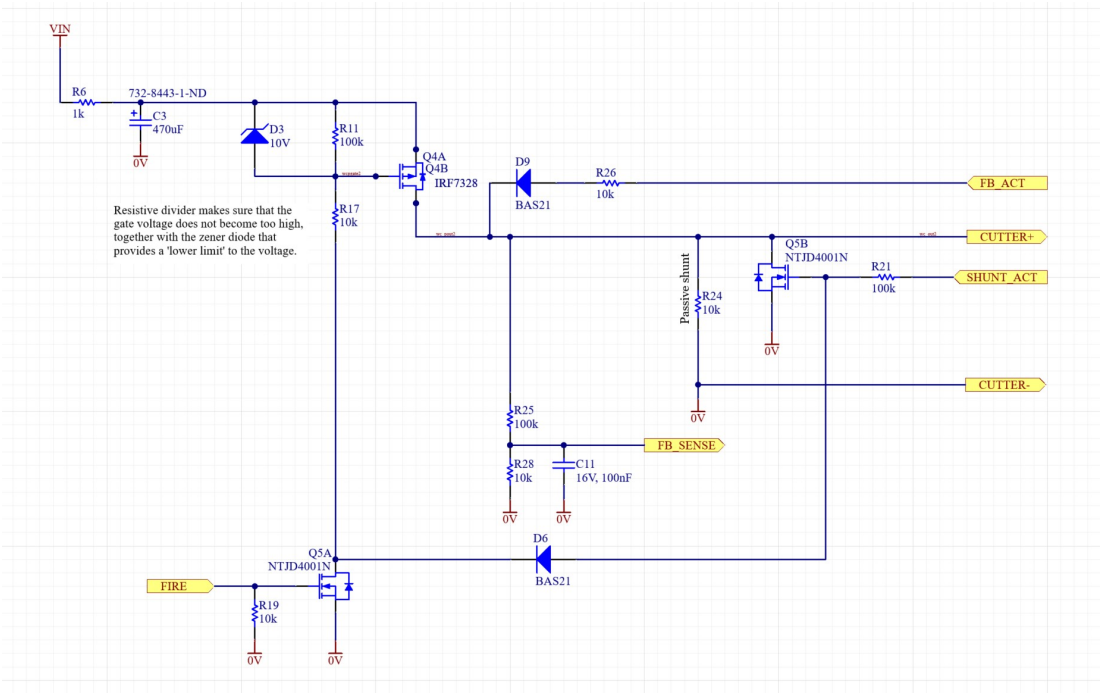


Figure 3: Cutter_driver schematic for Stratos IV

Principle of operation: C3 is slowly charged to VIN through R6, with a RC time constant of 0.5 seconds, C3 is almost fully charged (95%) after 1.5 seconds. This isolates the battery from the high current pulse. Q4a/b, a dual IRF7328 P-channel power mosfet is the main current switch (both mosfets are combined in parallel for higher switching capability). For protection, Q5B, an N channel mosfet, is used to shunt the Cutter+/- lines together.

When a positive signal on the FIRE input is sent, the shunt is disabled (gate pulled to ground), while Q4a/b is enabled, dumping the stored charge in the capacitor through Cutter+. For continuity testing, a 3.3V high signal is sent to FB_ACT by the microcontroller and read by an ADC at FB_Sense. No pyro results in 0.22v at FB_sense, while a pyro acts as a shunt leading to 0v at the sense pin (note that the shunt mosfet is disabled by software during this continuity checking). The sense current is approximately 0.26 mA.

Notes: Using a capacitor means there's no risk for browning out the power source, but these capacitors can be quite bulky. This circuit also doesn't contain arming mosfets (though this could be located somewhere else).

6.1.3 Spear

The spear electronics board used a darlington transistor with a shunt resistor and feedback in order to limit the current to 2 A. This approach works well, but during continuous actuation, may overheat due to the lack of over-temperature protection.

6.2 Requirements

- Capable of handling 2s (min 7v) as power source
- Capable of delivering 2A of current current for 0.x seconds (
- Idle test current of max 0.5 mA
- Can survive deliberate or accidental short circuit of pyro output indefinitely
- Software bugs should not be able to cause hardware damage
- Detection of physical arming switch status

Figure 4: ematch specs

- Detection of pyro continuity
- Allows digital arming (e.g. breakwire)
- Must be able to deliver the specified trigger signal with a failure rate of less than 1%
- Components resilient to vibrations and shock
- Allows self-check for key failure modes
- Interfacing with a 3.3v MCU (5v is rare for new MCU's)

Nice to have:

- Component count and bill of materials should be kept small for ease of design/assembly
- Component cost should be kept low if possible.

6.3 Thermal considerations

Mosfet self turn on https://toshiba.semicon-storage.com/info/application_note_en_20180726_AKX00074.pdf?did=59473

Thick power traces

6.4 Wonky pyro circuit

Sooooooooooo here's an idea... LM317 supports internal current limiting and over temperature protection. The current limit is between 1.5A and 2.2 A for a LM317T from STMicroelectronics (other vendors have similar values), which is perfect for a pyro actuator.

Suppose 8.4 Volts (2s pyro) is the input, and the 1 ohm E-match is the load. Suppose 2 amps is delivered by the LM317. This gives 2 volts across the E-match, and 6.4 volts over the LM317, which dissipates 12.8 watts. Suppose the delta temperature limit is 100 degrees.

TO-220 have a junction to air thermal resistance of 70 C/W. If left by itself, it'll thus heat up to 12.8*70=896 C. That's no good. But this doesn't happen instantly, the TO-220 has some thermal capacitance (heat capacity) equal to about 0.54 J/C[12]. The thermal time constant is thus 0.54*70=37.8 sec. Treating this as a RC circuit, the temperature over time is shown Figure 5. This indicates that the regulator is expected to overheat after approximately five seconds of continous pyro triggering. In real life, the E-match ignites within a fraction of a second. Overall, this indicates that thermals won't be an issue in normal operation, and that the thermal protection will kick in after approximately five seconds of short circuit fault.

In software, the pyro firing time will probably be limited to a second or so.

7 Electronic failure modes and mitigation

This section contains considerations about how the electronics may fail during either testing or on a flight, along with potential mitigation strategies to either avoid the problem, or ensure adequate data for a failure investigation.

7.1 Momentary power loss

During the flight, power to the electronics may be momentarily lost as a result of brownout, bouncing power switch, or bad connections to the battery.

Mitigation: Decouple the power electronics from the control, and have enough capacitance to survive the momentary power loss. Avoid shock-sensitive switches and use proper connections. If the microcontroller is reset during a longer power loss, it will be in the startup state, and

[12] https://www.onsemi.cn/pub/collateral/an-7516cn.pdf

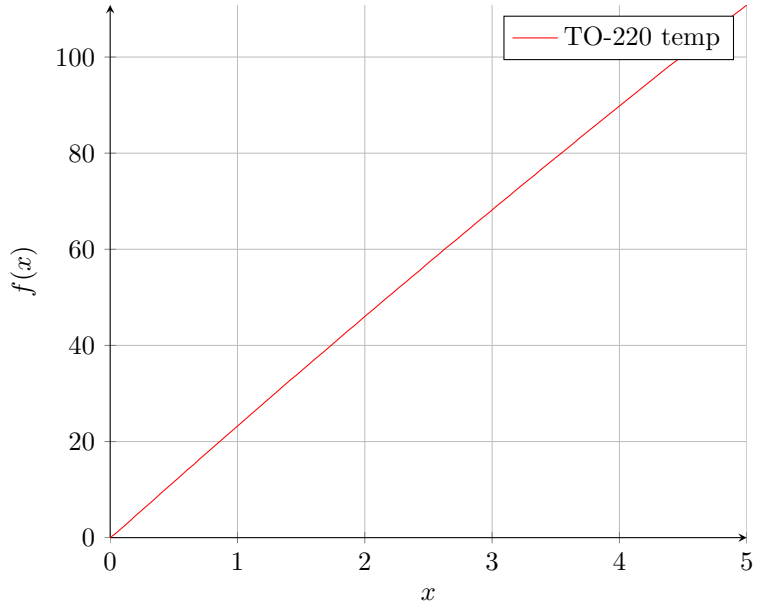


Figure 5: Estimated temperature vs time during continous pyro operation

likely not deploy recovery. Consider beginning data logging right away if system state indicates such a scenario.

Indication: A momentary drop in the battery readout may be read in the flight data. If reset, blackbox data may show the system tick timer going back to zero.

Result: Control electronics are hopefully kept active. Otherwise there should still be flight data.

7.2 Shorted pyro lead

If during the assembly, the leads to the pyro are shorted, this can cause a failure to deploy the recovery, since the leads are effectively shunted. This can also lead to damage to the high current circuitry.

Mitigation: Care should be taken to not have the E-match leads touch each other, both before and after actuation. Current limiting and over-temperature protection is used to avoid damage to electronics

Indication: This is indicated by the pyro actuation voltage staying low during the actuation, as opposed to having a quick downward spike.

Result: E-Match actuation may fail, but there should still be flight data to find the root cause.

7.3 Loss of power during flight

Due to shock or bad connections, the connection to the battery may be completely lost. Loss of power may also occur as a result of faulty or overloaded electronics causing a continous brownout. Depending on the stage during the flight, this may lead to an unsuccessful recovery.

Indication: Loss of power at early stages of the flight is indicated by a lack of recovery deployment, and sudden cut in both telemetry and blackbox data (if recovered)

Mitigation: Larger systems can use redundant power sources, but that doesn't fit here. Instead, avoid shock-sensitive switches, use proper connections. On the board side, the 3.3V supply is prioritized (directly supplied from the battery) with few failure-prone components.

Result: The chance of complete power loss is minimised with the mitigation strategies, but if

it were to occur, recovery would likely not deploy. Telemetry and/or flight data could be used to deduce this failure mode.

7.4 Reverse battery polarity

The user may accidentally plug in the battery in reverse, potentially damaging the electronics.

Indication: The electronics doesn't turn on, and may burn if not properly protected.

Mitigation: Use polarized battery connectors, and double check the polarity during assembly. Reverse voltage protection should be implemented on the circuit board.

Result: It just doesn't turn on.

7.5 Rocket explodes

The rocket may explode during the flight, potentially damaging the electronics.

Indication: Boom

Mitigation: Don't make rocket explode

Result: There should still be telemetry data until the moment of failure.

7.6 Launch detected too early

The flight computer may detect a launch event even if it hasn't occured, due to faulty wiring or sensor readout.

Indication: Early deployment of recovery.

Mitigation: Debouncing should be implemented on e.g. the breakwire, and acceleration thresholds should be tuned properly. For safety, audible cues should indicate the current flight state, alerting anybody nearby. The state machine should be extensively bench tested.

Result: The likelihood of faulty launch detection is minimised.

7.7 Launch not detected

Depending on the launch detection method, it may fail to properly detect the launch of the rocket, leading to the recovery not being deployed properly.

Indication: No deployment of recovery. Flight state machine never reaches the flight state, as indicated in both telemetry and blackbox data.

Mitigation: The launch detection method should be verified using both simulations, practical tests, and validation using previous flight data. If possible, the board can be used as a passive payload on a mission to validate the data from the actual hardware.

Result: If a breakwire is used, this shouldn't be an issue, but acceleration-based launch detection should also be reliable with proper validation (see e.g. almost every single commercial flight computer).

7.8 Pyro fires, but still has continuity

Found this video showing https://www.youtube.com/watch?v=YPxrc7PPgfw that pyros may still have continuity post firing, especially at lower currents when the resistive wire remains intact. This could cause unexpected current drain. If both a pyro and a servo are used, the servo won't have power during pyro actuation.

Indication: Successful pyro actuation, but continuity still detected. If pyro retries are activated, may retry a couple of times before giving up.

Mitigation: Current limiting, ground testing to see the behavior of the system in short-circuit scenarios. Limit the pyro actuation time to e.g. 0.5 seconds.

Result: Actuation worked, so hopefully everything went fine.

Servo jamming

Servo or pyro wires disconnecting during launch

8 Development plan and preliminary hardware tests

The following is a rough super optimistic plan for what needs to be done in terms of development.

- Finish designing hardware (7 days)
- LM317-based pyro circuit. Test max current, input voltage range, max actuation time, reliability, temperature (2-3 days)
- Get prototype board (10-15 days)
- Implement driver for Baro, IMU (1 day)
- Implement data logging, USB data transfer (1 day)
- Implement and validate flight state machine (1 day)
- Implement Pyro and servo actuation (1 day)
- Test pyro actuation using dummy loads. Test reliability. (1 day)
- Test pyro actuation with actual ematches. (1 day)
- Implement telemetry and "ground station mode" to send data to PC (4 days)
- Implement PC software for decoding and data storage. E.g. save to CSV (2 day)
- Implement NMEA GPS readout (1 day)
- Implement realtime data visualisation of Altitude, GPS location, acceleration, angular rates. (Try existing software) (4 days)
- Power consumption measurements (1 day)
- Telemetry range testing (2 day)
- Full long duration integration test (1 day)

Assuming some software development can be completed while waiting for shipping... Estimated development time is about 4-8 weeks. At the time of writing, the next scheduled launchday is in 4 months, so that should be fine :)

8.1 Pyro circuit testing

Due to the use of an unusual current limiting circuit, it is extensively tested to make sure it actually works. In order to conduct this test a "lab" power supply was built and used. Initially, a qualitative test was done with the lm317 voltage regulator shorting the supply rails. The observed behavior was an a current limit of 2-3 amps, followed by a drop after approximately five seconds due to the over-temperature shutdown. In the temperature shutdown state, a current of 0.4 amps still flows.

An improved test setup is used to get more quantitative data of the pyro ignition circuitry. The purpose of this test is to (1) get an accurate measure for the expected thermal shutdown time, and (2) get information about the behavior after repeated actuations. The test setup is as follows:

- Lab power supply with 4A current limit
- Arduino connected to a laptop for data acquisition
- 1 ohm power resistor to simulate the pyro and for current sensing
- Multimeter for low rate voltage measurements

Sooo ehh, no 1 ohm power resistor is currently available, so a long wire will be used as the shunt resistor instead. The wire is first folded over itself (avoid voltage spikes from inductance),

then coiled into a loop. Don't worry, it's Electroboom approved[13]. Running 1 amps through the shunt gives 0.84 volts and 2 amps through the shunt gives 1.7 volts (four wire kelvin measurement), giving a resistance value of approximately 0.85 ohms. This increases slightly with increasing temperature, but exact accuracy is not really important for this test, since we're only interested in rough values and the timings.

Two ADC channels are set up on the arduino. One to measure the system voltage, and the other to measure the current through the simulated E-match. This is sent at about 1000 Hz through the serial terminal to a laptop. The schematic for the test setup is shown in Figure 6. Due to the lack of the resistor values during assembly R3 and R4 were replaced by a potentiometer, with the output voltage adjusted to 5V.

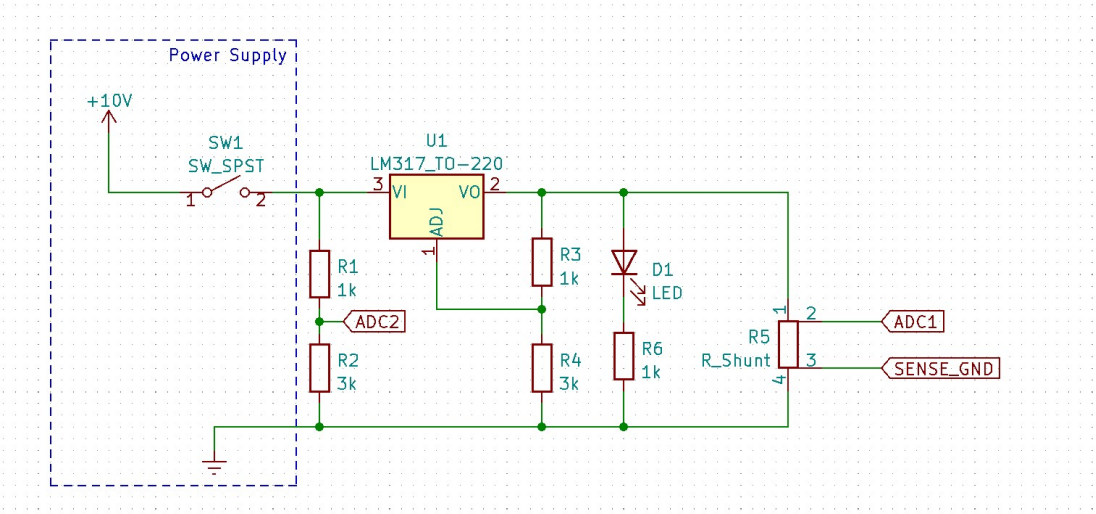


Figure 6: Pyro actuation circuit test setup

ADC1 measures the current through the simulated pyro, while ADC2 measures the input supply voltage. Furthermore, the regulator is programmed to output 5V, and an LED is added to indicate voltage on the output line. This circuit is built on a perf-board and a picture of the complete test setup is shown on Figure 7. Note that the test circuit is connected directly to the power supply with short wires since it was found that the available cheapo banana wires had a resistance of 0.9 ohms, thus messing up the input voltage, should really get some better ones...

[13] https://www.youtube.com/watch?v=j4u8fl31sgQ

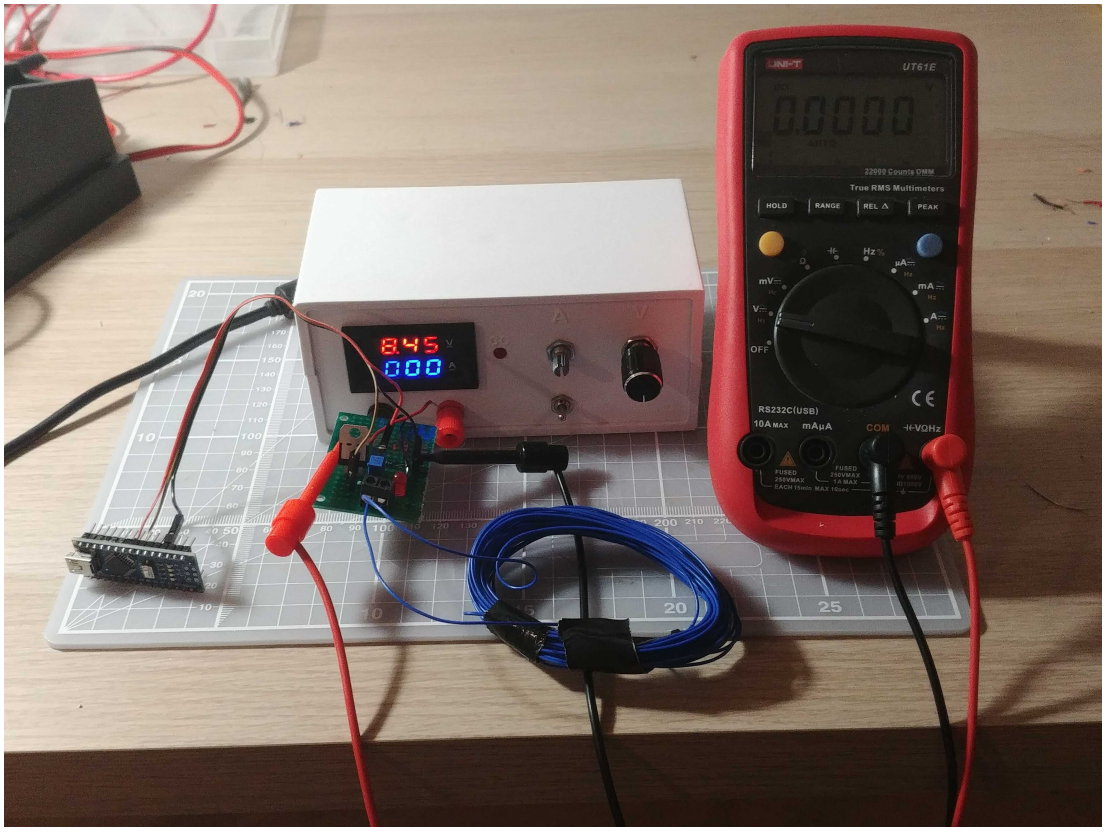


Figure 7: Pyro actuation circuit test setup

8.1.1 Max actuation time

For the initial test, multiple maximum-length actuations are performed as follows. The switch is turned on until the LM317 thermal shutdown as indicated by the LED while recording data. The data is then saved. When the LM317 is cool to touch again, the test is repeated. This is done a total of 5 times. An input voltage of 8.4 volts is used to represent a charged 2s lipo. The results are shown in Figure 8. During the plotting process, the initial current spike is used as time reference in order to align the data. The results indicate that the actuation is remarkably consistent, with the over temperature protection triggering at 4.2 seconds from actuation. Furthermore, the actuation current initially has a short spike at 3A, followed by a decrease to about 2A before the protection features kick in. This is perfect for actuating E-matches. After the protection circuitry has activated, short pulses of 2.3 amps are sent, which is indicated by the opaque block on the chart.

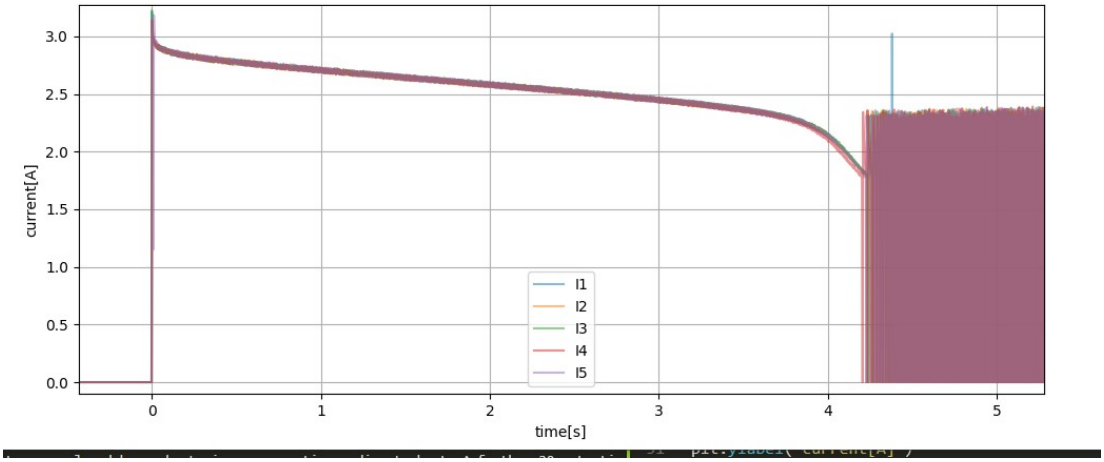


Figure 8: Current graph with 8.4V input

This is once again repeated with an input voltage of 7 volts, representing an almost discharged 2s LiPo with the results shown in Figure 9. Due to the lower input voltage, less power is dissipated in the regulator, leading to a longer time before overtemperature protection is activated. This occurs at around 6.2 seconds.

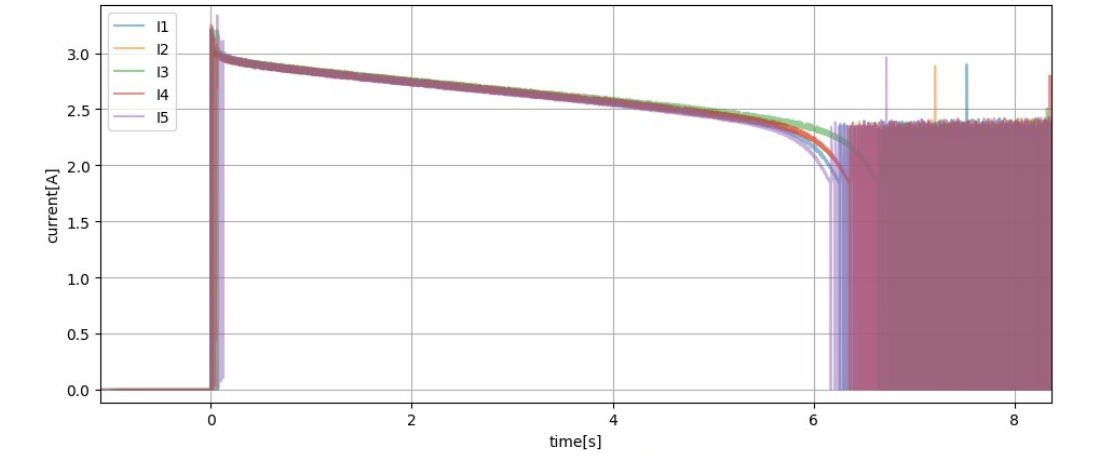


Figure 9: Current graph with 7V input

8.1.2 Reliability and stress testing

The previous actuation is performed continuously for 20-30 minutes. This serves as a rough way to stress-test the LM317 in current-limited to see if the behavior changes after a high number of cycles.

Firstly, continuous activation for 200 seconds was used to see how the current limit changes over time. This indicated that the current settles at about 0.5-0.7 amps.

Further stress testing was done by randomly turning the circuit on and off for 20-30 minutes, with over 100 actuations. Afterwards, the 0.85 ohm shunt was replaced by a short wire representing a direct short. A further 20 actuations were performed with this shorted configuration.

After a bunch of stress testing, let's do another actuation with 8.4v input from the cold state to see if the behavior has changed from the initial tests. A comparison between the initial

current graph and the current graph after stress testing is shown in Figure 10. Two identical tests were conducted after stress testing. One of which went into overtemperature protection 0.1 seconds before the other. Taking variations of the ambient temperature into consideration, there is essentially no difference in the performance compared to before the stress testing.

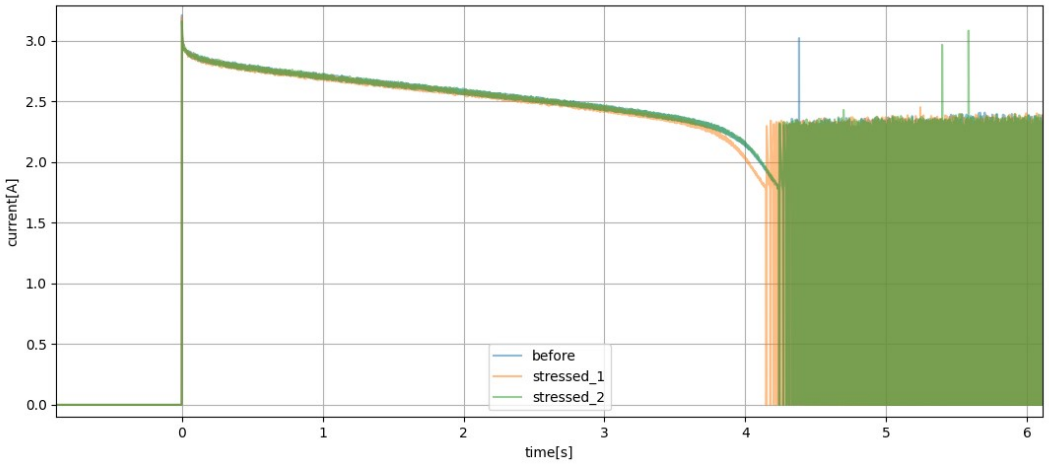


Figure 10: Results before and after stress testing

8.1.3 General results

The results from the test indicate that the current-limiting circuit based on the LM317 works well and very consistently. Based on the bench testing, it should be almost impossible to break the circuit by overcurrent or overheating. From room temperature, the total actuation time with a fully charged 2s lipo is approximately 4 seconds. Looking at the E-match specifications on Figure 11 the recommended actuation time is 150 milliseconds. In order words, there's plenty of margin. Overall, this approach is looking pretty good. Stress testing indicates no significant change in behavior after more than 100 anomalous actuations (pyro continues to draw current), a significant improvement to circuits without current/thermal limiting, which may experience thermal runaway and fail from one anomalous actuation.

Figure 11: ematch specs

8.2 Telemetry range testing

Details will be added here

9 Schematic design

For the schematic design, the RP2040 hardware design guideline is used as reference for wiring up the rp2040 MCU[14].

Turns out the RP2040 only has 4 ADC pins, so they will be used for:

- VBAT voltage
- Channel 1 continuity
- Channel 2 continuity
- Actuator sense

10 PCB Design

10.1 Design rules and stackup

Four layer impedance controller PCB from JLCPCB. JLC7628 or JLC3313. They have an impedance calculator[15].

[14] https://datasheets.raspberrypi.com/rp2040/hardware-design-with-rp2040.pdf
[15] https://cart.jlcpcb.com/impedanceCalculation

The capabilities of interest[16] for 4-layer boards:

- +/- 0.2mm CNC routing tolerance
- Drill hole size: 0.20-6.30mm
- Min via hole size: 0.2mm
- Min via diameter: 0.45mm
- Min nonplated hole size: 0.50mm
- Min annular ring size: 0.13mm
- Min pad to track: 0.2mm
- 3.5mil (0.09mm) trace width and clearance
- 5mil via clearance
- pad-to-pad clearance (no holes): 0.127mm

Adding a bit of margin, the following are used for the design.

- Signal via 0.25mm hole, 0.50mm via size (tented)
- Power via: 0.4mm/0.7mm
- 0.15mm signal trace
- 0.3mm supply trace
- 1.0mm or higher high current trace

10.2 Silkscreen

This subsection contains info about the stuff on the silkscreen.

Revision number: Revision number in MAJOR.minor format (e.g. 1.0, 1.1, 2.0). The major version indicates significant hardware/wiring changes which might require configuration differences in software. The major revision should be incremented if e.g. a sensor is swapped out or if the size of the board/connectors are changed. Boards with the same revisions should be able to run on exactly the same software configuration and have the same physical dimensions, meaning one board can be swapped with another board in a rocket without any changes.

The minor revision number indicates changes in the board, which don't affect either software or hardware integration. This could be bug fixes, changes in routing, or changes in the passive components (voltage regulators etc). If any significant bugs are found, these should be documented.

Date/Batch number: A number indicating the PCB/SMT batch. After being sent for PCB and SMT assembly, the design is manually reviewed and modified (DFM review), so this keeps track of which batch the PCB is from. Since a date is nice to have the date as well, the date at which the design files are generated are used. This date also serves to uniquely identify the PCB batch.

Polarity/pinout indicators: Important pin/connectors should be marked

Serial number field: A field where the serial number can be written

Project name/logos: Also add the DARE logo

10.3 Layout/routing

In order to figure out the layout, the components were placed into the PCB editor in Altium and rearranged in order to figure out a nice placement. This part is kinda hard to document since it involved quite a bit of messing around to squeeze things into the contrained space.

Initially the screw terminals were placed on the backside of the board, afterwards it turned out that there was space on the top side after all. The back of the PCB contains the bulky components, namely the buzzer, the 3.3V voltage regulator, and the LM317.

[16] https://jlcpcb.com/capabilities/pcb-capabilities

In general, components were placed in neat rows.

Since it's going to be assembled, component designators are hidden.

high speed and important signals (e.g. clock and QSPI) are prioritized. high current traces are 1-2 mm etc.

Medium-speed signals are routed next. This includes the chip select and PWM pins. Finally, the low speed signals (LED etc) are routed between everything else.

10.4 PCB revision history

10.4.1 1.0

This is the initial version to be manufactured.

10.4.2 1.1

- Fix RF transceiver silkscreen location
- Fix incorrect power trace width

11 Ordering the PCB/SMT

When ordering the board, the following steps should be performed.

- Get the newest project files
- Use the default output job to generate the Gerber, drill, BOM, and pick/place files. This updates the date timestamp
- Move the drill files to the gerber folder, and put everything in a zip folder.
- In the JLCPCB order form, upload gerber zip
- (Add screenshot of order settings)
- Select SMT assembly. Top side only. Tooling holes: added by customer
- Upload BOM and Centroid files
- Check parts placements

12 Final cost and PCB/component ordering

Black or purple would've been neat, but apparently the economic PCB manufacturing option only allows green[17].

12.1 Other components

Other components that need to be ordered:

- u.fl connectors (?) x 10
- SMA connectors x 10
- 2.54mm screw terminals (2 pin and 4 pin) x 20
- LM317T in TO-220 package x 10
- 2.54mm pinheaders x 20
- SMA rubber ducky antenna x 5
- Loud buzzer x 10
- W25Q128 flash chip (extra blackbox, optional) x 10
- M2 mounting screws + heat set inserts x ??

[17] https://jlcpcb.com/capabilities/pcb-assembly-capabilities

-

Antenna options (no ground plane needed):

- https://www.lcsc.com/product-detail/Antennas_B-T-AG-010518-0818_C2875301.html
- https://www.lcsc.com/product-detail/Antennas_ZIISOR-TX868-JKD-20_C468326.html
- https://www.lcsc.com/product-detail/Antennas_BAT-WIRELESS-BW868JWX105-10KJ_C496578.html

Other useful things:

- UF.L to SMA: https://www.lcsc.com/product-detail/Radio-Frequency-Cable_BAT-WIRELESS-BWIPX1-C784405.html
- SMA extension cable: https://www.lcsc.com/product-detail/Radio-Frequency-Cable_ZIISOR-XC-SMA-SMA-20_C2693325.html

Prelim shopping cart using LCSC (Enough parts for 10 boards):

- 10x U.FL connectors: https://www.lcsc.com/product-detail/RF-Connectors-Coaxial-Connectors_HRS-Hirose-U-FL-R-SMT-1-10_C88373.html
- 10x edge-mount SMA: https://www.lcsc.com/product-detail/RF-Connectors-Coaxial-Connectors_BAT-WIRELESS-BWSMA-KE-P001_C496550.html
- 10x LM317: https://www.lcsc.com/product-detail/Linear-Voltage-Regulators-LDO_STMicroelectronics-LM317T-DG_C18718.html
- 5x LM350 (For higher power servos): https://www.lcsc.com/product-detail/Linear-Voltage-Regulato HGSEMI-LM350T_C512787.html
- 20x 2 pin screw terminals: https://www.lcsc.com/product-detail/Screw-terminal_DIBO-DB125-2-54-2P-GN-S_C918120.html
- 10x 4 pin screw terminals: https://www.lcsc.com/product-detail/Screw-terminal_DIBO-DB125-2-54-4P-GN-S_C918122.html
- 10x 4 pin JST terminals: https://www.lcsc.com/product-detail/Wire-To-Board-Wire-To-Wire-Connec JST-Sales-America-B04B-PASK-LF-SN_C265078.html (We should already have these)
- 10x loud buzzers: https://www.lcsc.com/product-detail/Buzzers_KINGSTATE-KXG1205C_C1122355.html
- 5x long antennas: https://www.lcsc.com/product-detail/Antennas_ZIISOR-TX868-JKD-20_C468326.html
- 5x short antennas: https://www.lcsc.com/product-detail/Antennas_ZIISOR-TX868-JK-11_C468325.html
- 1x cheap GPS module (testing/dev): https://www.lcsc.com/product-detail/Satellite-Positioning-M Vollgo-VG7669T160N0MA_C5197313.html
- 10x 9x2 pinheaders: https://www.lcsc.com/product-detail/Pin-Headers_XFCN-PZ254V-12-18P_C492426.html
- 10x W25Q128 flash chip (extra blackbox): https://www.lcsc.com/product-detail/NOR-FLASH_Winbond-Elec-W25Q128JVSIQ_C97521.html

Total (excluding GPS module): 37.00 EUR. So extra components come out to under 4 EUR per board. Looking at LCSC for now since they have cheap antennas etc.

13 Assembled PCB's and hardware testing

An initial batch of five PCB's including assembly were ordered on the 9th of january, 2023. About 10 days later on the 19th, the boards were received. Visual inspection of the rev1.0 boards revealed the following:

- Actuator channel 2 power trace was accidentally reduced in width, limiting the maximum current handling capability.

- The 78m05 regulator positioning can be tweaked a bit.

Functionality wise, the boards seem to behave as expected, with the communication between the rp2040 and the sensors/transceiver working as expected. Furthermore, the L80Re gps module was tested by connecting it to a USB-serial adaptor and a smartphone. In urban environment, the GPS accuracy was about +/- 20 meters, which is not great, but fine for recovery purposes.

13.1 Power consumption

The following are some power consumption measurements. Since the board only has linear regulators, supplied current is also used by the individual IC's.

- L80-R GPS module: 24 mA
- Kolibri idle/unprogrammed: 20mA.
- Kolibri low power continuous wave transmission: 80mA.

14 Firmware development

The plan is to write the software in the pi pico C/C++ SDK. The main reasons are speed, static typing, and support for debugging. It also supports micropython/circuitpython which is easier to use and set up but ehh. In the interest of keeping things clean and speedy, C/C++ it is. All the configuration should be possible without modifying the firmware, but if someone wants do something else (e.g. high speed data logging for a ground test), they can prototype something in micropython.

Firmware development checklist

- Set up development environment (DONE!)
- USB Serial communication and LED blinks (DONE!)
- SPI Driver for BARO (DONE)
- SPI Driver for IMU (DONE)
- Driver for ADC
- Driver for selectable actuation channel
- Persistent serial number
- Encoding and decoding of blackbox binary format
- Servo PWM driver (DONE)
- Base SPI driver for rf transmission (check continuous tone with SDR) (DONE!)
- SPI Driver for transmitting/receiving LoRa packets. (DONE) Document modulation parameters
- SPI Driver for transmitting/receiving FSK packets. Document modulation parameters
- Implement persistent key-value variables for settings
- Set up (notion?) database as documentation and operational history for each serial number. (Will contain flight history, data, and repairs/modifications) (DONE)
- Write driver for MTK gps module and NMEA parsing
- Write basic flight state machine based on SRP (DONE)
- Task scheduler (done)

14.1 Programming guidelines

It would be nice to implement the software in an object-oriented manner, which should be easy to read. E.g. pseudocode:

```
baro = LPS22.new(i2c1)
baro.init()
```

```
while True:
    baro.read()
    pres = baro.pressure()
    temp = baro.temperature()
```

None of the function calls should have the capability of getting "stuck" in a loop. In other words, if a blocking functionality is implemented, it must have a defined timeout. Look into implementing a watchdog timer. The following are some other notes based on *Fault-tolerant systems* by C. Mani Krishna and Israel Koren (chapter 5):

- Acceptance checks: Detect if measured values and timings are reasonable. E.g. error if outside defined bounds.
- Avoid buffer overflows with proper buffer lengths/checking.
- Wrappers may be used around existing code to ensure correctness.
- Perform slightly different redundant calculations and compare (Not relevant)
- N-version programming. Same software by multiple teams, and vote on output (Yeah... nah)
- Diversify both hardware and software. E.g. main implementation with backup if main fails (Not relevant)
- Proper exception handling if relevant.

Not really relevant, but here's a fun model from that book: *MUSA-OKUMOTO MODEL: This model assumes an infinite (or at least very large) number of initial bugs in the software, and similarly to the previous model, uses M(t)—the number of bugs discovered and corrected during time [0,t]. (...) Under this model, the error rate after testing for a length of time t is given by:*

$$\lambda(t) = \lambda_0 e^{-c\mu(t)} \tag{1}$$

Always nice to have an infinite number of bugs :)

The pi pico does not have a floating point unit, so floats are pretty slow. According to the SDK datasheet (page 23), optimized standard functions are available in pico/float.h and pico/double.h. This includes stuff like conversion between floats and integers, and exponents. Since doubles aren't needed, let's use floats for everything.

Also, look into properly setting up darestl for the rp2040.

14.2 Hello world! and setting things up

Setting up the C/C++ sdk was kinda a pain on windows, but oh well. Using the pico stolen from Leo, the hello world and blink programs from https://github.com/raspberrypi/pico-examples were checked out. A helper script was made to automate the building and programming to the pico.

Since it might be a bit annoying to have to unplug power, hold down the boot button, and replug power each time, it turns out there's a way of rebooting into the bootloader mode automatically. https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/pico_stdio_usb/include/pico/stdio_usb.h#L47. The trick is to set up USB serial communication, and change the baud rate to 1200 bps (default), a magic number which indicates that the pico should go to boot mode. Pretty nifty. In order to test this functionality, a Python script based on Pyserial was set up, and it seems to work. By changing the baud rate to 1200, the pico resets and goes into bootloader mode. Hmmm... maybe we can remove the boot button after all? That would save a few more cents and square millimeters.

For setting up the project properly, this guide was followed: https://admantium.medium.com/getting-started-with-raspberry-pico-and-cmake-f536e18512e6

The darestl library was also included thus making it possible to use LBP etc. without having to rewrite it. Pretty neat.

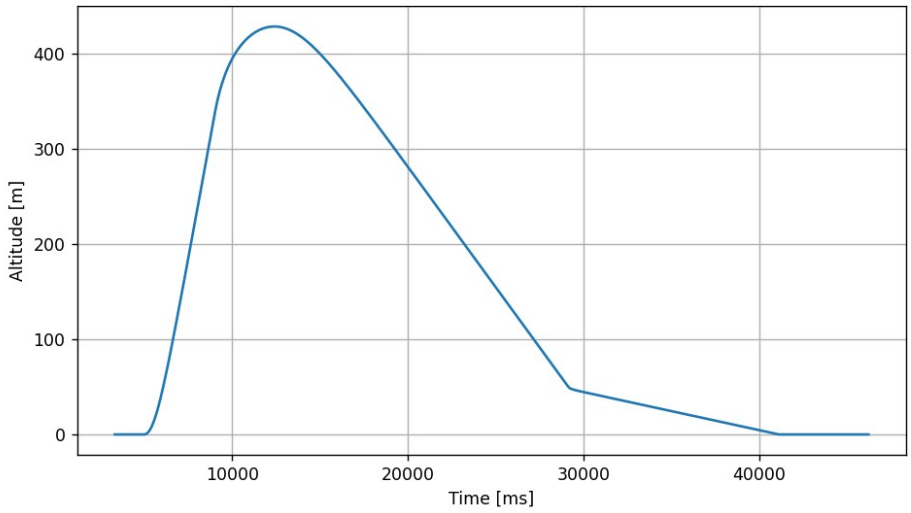


Figure 12: Initial result from 1D flight simulation

14.3 Documentation and unit testing

In order to hopefully not make spaghetti code, the code will be documented using inline comments and doxygen. This is useful for future people interested in working on the software.

In order to make sure the library code works as expected, a secondary program is set up to use the same header files. Not sure how extensive this will be, but it could be neat to include a simulation to check the state transitions/timing. But for now we'll just check the functions.

14.4 Flight simulation

A basic 1D simulation program was implemented in C++. This can serve to test the functionality of any altitude-based deployment. The initial program was pretty simple. It assumed constant thrust for a certain burn time, while also including a basic drag and parachute model. The altitude vs time plot is shown on Figure 12;

Now... would it be possible to run this on the second core? Possibly. The initial implementation is not particularly optimized and uses floating points extensively. Due to the RP2040 lacking a floating point unit, this simulation would run extremely slowly. Assuming an average floating point computation time of 800 nanoseconds[18] and 50 operations per cycle, this takes 40 microseconds. Considering the update rate of the barometer at under 100 Hz, this should be plenty.

14.5 Speed, task scheduling and multitasking

The firmware has to do quite a bit of stuff. So how to organize that? In order to keep timings neat, a minimum target loop time is set to 200 Hz (though this may change depending on performance margin). Since the pico has a second core, some tasks can be offloaded there. For now, let's focus on using just a single core.

Assuming the SPI communication to the barometer, IMU, and and transceiver occurs at 10 MHz. The rough time required for blocking data transfer are: DMA would be possible, but assume just polling for now.

The QSPI flash chip at 133 Mhz runs at about 532 Mbps.

[18] See page 24: https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf, seems like benchmark is for 100 computations based on https://forums.raspberrypi.com/viewtopic.php?t=308794

The GPS communication occurs through UART to a FIFO buffer at 115200 baud, so that's should be non blocking. In any case, a 70 byte NMEA string takes 4.86111111 milliseconds to read.

- IMU: 16*6 bits, 9.6 microseconds
- Barometer: 24 (pres) + 16 (temp) bits, 4 microseconds
- Transceiver, 32 byte packet, 25.6 microseconds
- Blackbox write page, 256 bytes, 3.849 microseconds

Total: about 45 microseconds. So if the loop time is set to 1 Khz, there's still 950 microseconds to do other stuff. Not too bad :). By comparison, if IMU and barometer communication were through 400KHz I2C, the transfer time requirements would be about 400 microseconds. For the implementation, might bump the SPI speed down to 5 MHz for reliability (though traces will be very short, so unlikely to be an issue).

If we go for 1000 Hz loop time, each iteration could be something like:

- Update state machine
- Read and process ADC data
- Read and process IMU
- Read and process baro
- Read GPS and process GPS
- Encode telemetry packet
- Write telemetry packet
- Encode blackbox packet
- Write blackbox package
- Sleep for remaining loop time.

Each read operation only needs to be performed if there's new data. Similarly, the transceiver and blackbox only needs to be used intermittently with a certain sampling rate. The sleep cycle ensures that the behaviour remains consistent time-wise even with the addition of new code.

For the data processing, there's still quite a bit of processing power remaining. Based on the rough estimates, a 1000 Hz loop time should be fairly doable.

Both the IMU and Baro have output pins indicating the presence of new data, so that's nice.

Due to laws and stuff, the RF transmitter cannot transmit continuously. It could thus be nice to switch it to receive mode during the waiting intervals, thus making two-way communication possible.

Multitasking systems are either preemptive or non-preemptive. Preemptive operating systems (Typically real time operating systems or RTOS[19]) allow for multiple tasks running concurrently, with near instantaneous context switching if a different thread has higher priority. This is advantageous for real-time systems with strict timing requirements but has a very high implementation complexity, though there are existing options like FreeRTOS. As noted in the Altus Metrum AltOS documentation[20] typical amateur rocket control systems don't really have strict timing requirements (doesn't matter if the parachute is deployed 1 ms later), so it can be easier to go for a non-preemptive scheduling system. AltOS uses some ARM assembly code to handle the context switching and state preservation. For the initial version of the scheduler for this project, "proper" context switching won't be implemented. The downside is some speed penalty, but based on the previous calculations, there's quite a bit of processing power available. This also makes it possible to simulate and validate the code on the PC side, since there isn't platform-specific assembly code. Other places to take inspiration from are open source multicopter flight control firmwares like betaflight[21] which are fairly readible.

[19] https://en.wikipedia.org/wiki/Real-time_operating_system
[20] https://altusmetrum.org/AltOS/doc/altos.html
[21] https://github.com/betaflight/betaflight/blob/master/src/main/scheduler/scheduler.c

Note that non-preemptive or cooperative multitasking requires that the tasks willingly "give up" control when they are done. During the development of the tasks, they must thus have to adhere to certain timing constants while avoiding blocking functionality or infinite loops. Since the scope of the tasks we are dealing with are limited, this should be fairly easy to handle.

For the tasks, an base C++ class is defined, with virtual methods for the init and update functions. It also includes useful parameters to handle a fixed refresh rate. For an overview on non-preemptive and preemptive scheduling, see this video: https://www.youtube.com/watch?v=4DhFmL-6SDA. For sharing data across different tasks, we could define a global struct to contain all the available values. With the non-preemptive system, there shouldn't be major issues with race conditions, since each task is in a known state while other tasks are executed. With this approach, it might be tricky to handle precise delays etc. but hopefully that isn't really a requirement. If the main task scheduler runs at 1000 Hz, the timing accuracy of each task is expected to be within 1 ms. Honestly that's fine.

In order to evaluate the performance of the tasks, the base class also includes functionality to time the execution time. This allows for predicting the worst case CPU load (e.g. all tasks in one cycle), also known in the biz as utilization. In the event of a failure within one task, a watchdog timer can reset the execution. Care should be taken to avoid this. The priority of the tasks using this timed round-robin approach is essentially the order of the task execution. it's also possible to skip tasks to the next cycle if timing becomes a constraint later on. Here's a nice video showing how round robin schedulers can be implemented: https://www.youtube.com/watch?v=0H8wDxMBl2o

Speaking on timing, what format do we want to handle the timestamps? For now, let's use unsigned 32 bit integers representing the microseconds since boot. This overflows after 1.19 hours, but the time delta computations should still give the correct results due to how unsigned integers work. Incidentally the same time format is used for the Sparrow/Stratos 5 labview code.

14.6 LPS22 barometer driver

This one is fairly simple to set up. Let's just use polling mode (no FIFO) for now.

Setup:

- Configure with CTRL_REG1. Set data rate to 75 Hz (ODR=101) without LPF, continous update: 01010010

The bandwidth without LPF is sample rate/2 (i.e. nyquist). We'll see how noisy this is... but it's probably nice to get the raw data for now. For actual altitude-based deployment, some filtering is needed.

Read data

1. Check if data is new: Either using pin or reading STATUS register.
2. Read registers 0x29, 0x2A, 0x2B (H,L,XL bytes)
3. Combine to one 24 bit number. Treat as signed integer
4. Scaling is 4096 LSB/hPa. Scale to get pressure.

Not sure why it's signed, since that allows negative pressure, but whatever.

The altitude can be computed based on the ISA formula.

14.7 GPS development

With the MTK-based GPS modules, the configuration is handled with the PMTK commands. Refer to the manual https://auroraevernet.ru/upload/iblock/44c/44c84ca49922accf2dbee06c51c3490c.pdf. Each PMTK command ends with a checksum (XOR of all characters between $ and *). The checksum can be found with this simple python script:

```
s = "PMTK251,115200"
out = 0
for ch in s: out ^= ord(ch)
print(hex(out))
>>> 0x1f


def checksum(s):
    out = 0
    for ch in s:
        out ^= ord(ch)
    print(hex(out))
```

Initially, the L80RE GPS has a default baud rate of 9600 baud. Set the baud rate to 115200: `$PMTK251,115200*1f<CR><LF>`

Set max output rate for RMC and GGA sentences:

`$PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*28<CR><LF>`

The MTK has different operating modes, normal, fitness, aviation or balloon mode. Balloon mode can apparently operate up to 80 km... Let's set PMTK886 to balloon mode for now... "In mode 0 2, the altitude limitation is 10,000 meter. For mode 3 the altitude limitation is 80,000 meters; however when the altitude exceeds 18,000 meter, the velocity must be lower than 515 m/s."

14.8 Launch detection

Accelerometer-based launch detection is typically used in commercial flight computers. The tricky part becomes finding the correct threshold values to prevent any false positives, while reliably detecting the launch. As an example implementation, let's have a look at the threshold values used in the Altus Metrum systems (AltOS[22]). Looking at `src/kernel/ao_flight.c`, here's the relevant part for launch detection:

```
1  case ao_flight_pad:
2          /* pad to boost:
3           *
4           * barometer: > 20m vertical motion
5           *              OR
6           * accelerometer: > 2g AND velocity > 5m/s
7           *
8           * The accelerometer should always detect motion before
9           * the barometer, but we use both to make sure this
10          * transition is detected. If the device
11          * doesn't have an accelerometer, then ignore the
12          * speed and acceleration as they are quite noisy
13          * on the pad.
14          */
15         if (ao_height > AO_M_TO_HEIGHT(20)
16 #if HAS_ACCEL
17             || (ao_accel > AO_MSS_TO_ACCEL(20)
18                 && ao_speed > AO_MS_TO_SPEED(5))
19 #endif
20                 )
21         {
```

[22] Source code available here: https://git.gag.com/?p=fw/altos;a=summary

```
22                  ao_flight_state = ao_flight_boost;
23                  ao_wakeup(&ao_flight_state);
24
25                  ao_launch_tick = ao_boost_tick = ao_sample_tick;
```

AltOS computes the speed based on both barometer and acceleration data using a kalman filter. AltOS also has some interesting non-preemptive os/scheduling code. Hmmmm..

14.9 Blackbox logging and persistent storage

So the plan is to log the flight data in the same flash chip used for the program. This might make it tricky to use dual-core operation... see this page: https://kevinboone.me/picoflash.html?i=1 and the api docs: https://raspberrypi.github.io/pico-sdk-doxygen/group__hardware__flash.html

This is hopefully fine for single core operation?? We'll see.

Using the pico sdk examples, the erase operation takes 35873 microseconds (one sector or 4096 bytes) while the write operation takes 588 microseconds (one page or 256 bytes). Due to the way flash works, only zeroes can be written, thus requiring a complete erasure before stuff can be written. Luckily the erasure can be done in advance. Since the flash memory is also used for execution, the write operation must be completely blocking, with no interrupts or anything allowed. If the target time resolution of the firmware is 1 millisecond, this blocking write should be fine?? Data logging occurs during flight, when so interrupts. On the other hand, using a separate storage medium can be considered to avoid this problem altogether but ehhh. (Note: at the time of writing, the hardware design isn't yet finalised). For now, this approach should be fine, especially at lower blackbox logging rates. If more storage is desired, the serial interface can be used to add external data storage. Reading data from the flash is really quick, and can be done in a non-blocking manner, so data readout shouldn't be a problem.

When writing data, care should be taken not to overwrite the program data. Flash should also be used to store the persistent data and settings. Let's suppose that the program memory isn't expected to reach more than 2 megabytes (the amount of storage on the official pico board). With some buffer, let's say that the configuration and logging data starts at 4 megabytes. The layout of the flash memory would be something like this:

```
MB|data
--------
00|START OF FLASH
--|offset
--|program memory
--|buffer
04|configuration memory
--|buffer
--|blackbox data
--|buffer
16|END OF FLASH
```

The configuration memory would include stuff like the deployment timer and recovery actuation settings. During startup, the segments of memory assigned to the configuration memory and blackbox data can be read very quickly (i.e. just looking at the first byte in each page). In pseudocode, the startup sequence could be something like:

```
1  read(configuration_memory)
2  if configuration_memory == blank:
3      # flash not initiated
4      erase(configuration_memory)
```

```
    erase(blackbox_data)
    write(default_configuration_memory)

else:
    # already initialized
    config = read(configuration_memory)

# check if blackbox has data:
for page in range(num_blackbox_pages):
    if read_byte(blackbox_page[page]) != blackbox_header:
        start_page = page
        # + some logic to check if the whole thing is full
        break

blackbox_filled_amount = start_page/num_blackbox_pages
```

If the whole thing is full, we can either start over from the beginning, or give an option to manually trigger flash erase. Starting over from the beginning would require a complete (slow) flash erase cycle. Since this is a very slow blocking operation, manual trigger is probably the way to go. The data readout should then allow the user to see how full the blackbox is in percents with an option to erase it. Since each flight is only expected to take 1-2 megabytes at most, this should probably be fine.

As a safety measure to prevent data loss, two blackbox logging rates can be defined. A default one (e.g. 100 Hz) and a slow one (e.g. 1 Hz) with the logging process switching to slow data logging when the blackbox memory is almost full. The slow data logging mode can also be enabled at all times since even 1 MB is enough for multiple hours of data logging. In order to avoid ambuguity of the logged packets, identifiers should be used to signify distinct logs (e.g. a numerical value in the header combined with the system time) this makes it easier to separate multiple flights saved on the same log.

If the whole configuration memory is to be updated, a sector erase should be performed followed by a page write. Since this is a user interaction thing, the 35 milliseconds of latency shouldn't me a big deal.

By using proper headers and addressing, the data can also be recovered by dumping and parsing the data from the flash chip in the event of a catastrophic failure.

14.10 SX1262 driver

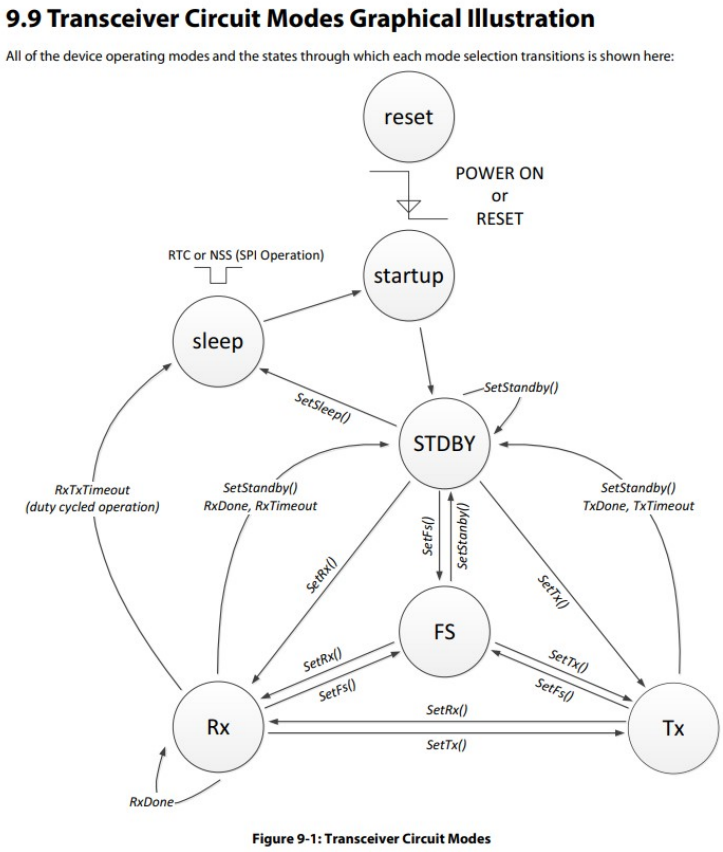


Figure 13: SX1262 internal states

The crystal on the sx1262 module is 32MHz 10ppm, which limits how narrowband the transmission can be. This section just contains some notes for configuring the registers of the radio transceiver for various tests/modes.

Tx in the 869.4 - 869.65 band is allowed up to 500 mW with a 10% duty cycle. This gives 869.525 center with a 250 KHz bandwidth

14.10.1 Tx test

(SetRegulatorMode(LDO): Default already)

In standby, module can be clocked from low power RC oscillator or crystal. Use crystal for tiny timing benefits (saves 120 microseconds of startup). SetStandby(XOSC): 80 01 SetPacketType(LoRa): 8A 00

Find rf frequency bits with RFfreq * Fxtal / (2$\hat{2}$5))INVERT FORMULA 869.525e6 * 32e6 / (2$\hat{2}$5) = 829243659.973 = 829243660 = 0x316D410C

SetRfFrequency(869.525): 86 31 6D 41 0C

Power and PA are set to low power PA and +14 dBm by default. To change: To high power PA, see section 12.1.14.1 in DS for optimal settings. For 22 dBm SetPaConfig(paDutyCycle, hpMax, devicesel=1, paLut): 04 07 00 01 Power is signed int tx power in dBm. To set the power to 0 dBm with 40 microsecond ramp time: SetTxParams(power, ramptime): 0x00 0x02

SetTxContinuousWave, 0xD1

We should now see a peak at 869.525 MHz on the SDR :)

Apparently there are options to trim the frequency: https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R0000001O4B/NlRzHNuT0dwHAFmCc4OezjPw9YrUW3k1Ghd5bZAy0m8. LoRa modulation is quite tolerant to frequency deviations, but this could be nice for FSK.

For super low power Pa config: First byte is opcode. Set Pa to low power max 14 dBm (no amplification): 0x95, 0x02, 0x02, 0x00, 0x01 (Default) Set Tx params to -17 dBm, 40uS ramptime: 0x8E, 0xEF, 0x02

SetTxContinuousWave, 0xD1

14.10.2 LoRa

14.10.3 FSK

FSK modulation allows for higher data rates, but doesn't do any error correction out of the box. It does do whitening and CRC checks, but adding a simple forward error could be neat in the future.

Let's configure it for a target data rate of 50 kbps. With 32 packets, this allows up to about 200 Hz packet rate. Let's just aim for 50 Hz for now, which leaves a bit of margin.

In FSK mode, pulse shaping, frequency deviation, and data rate are available. There's the bitrate setting:

$$BR = \frac{F_{XOSC}}{BitRate} * 32 \tag{2}$$

So for 50 kbps, BR has to be set to 20480

The frequency deviation is given by:

$$Fdev = \frac{FdevHz}{FreqStep} \tag{3}$$

$$FreqStep = \frac{XtalFreq}{2^{25}} \tag{4}$$

Shaping filters are applied to limit the bandwidth, which must follow (2*Fdev + BR) ¡ BW

Assuming 10 ppm frequency error at 868 Mhz gives 8.68*2 kilohertz. As previously mentioned, we're limited by a 250 KHz bandwidth by laws.

The filter bandwidth must be more than BR + 2 * FreqDev + FreqError.

Now the question is, what frequency deviation to use? If minimum-shift keying is used, the minimum frequency deviation is 50000/4 = 12.500 KHz. Selecting a multiple of this as the frequency decreases intersymbol interference [23]. So let's try FreqDev of 25 KHz (50 Khz difference between upper and lower frequency)

Fdev = 25000/(XtalFreq/2$\hat{2}$5) = 26214.4 = 26214

14.11 Flight state machine

The state machine for operation is based on the SRP board state machine with some modifications.

15 Ground station software

Soooo ehhh, this is not really needed, but it would be nice to have a nice dashboard for receiving the telemetry data in a pretty way. It receives the data over USB, and does some processing.

[23] https://www.sciencedirect.com/topics/computer-science/frequency-shift-keying

Required features

- Read data over USB
- Display raw data
- Decode data packets
- Save data to the disk

That's fairly simple to do, but what if we want more? Fancy features for the ground station software could be a dashboard with indicators for:

- Barometric pressure
- Flight state
- Actuator status (i.e. turn red when pyro is firing)
- Signal strength
- GPS coordinate on map
- Altitude (dial or bar chart)
- Velocity (Speedometer type thing)
- Gyro rates
- Acceleration

It could also be neat to do it in a way which enables easy use with a video stream to work as a telemetry overlay. The easiest way to allow for that would be to develop the graphics in HTML/CSS, but we'll see... There is some telemetry overlay code from Stratos IV which could be adapted for it.

Since telemetry packets are in different formats, it would be nice to have unified codebase to handle data encoding and decoding, so it's possible for the ground station visualisation software and the transmitter to use the same definitions, but we'll see...

The fancy visualizations can also be done using existing software. Possible choices are:

- DARE-MCS
- Stratos Java client
- Serial Studio

Looking over DARE-MCS, it seems pretty neat and could be a nice too to handle the data visualization without having to write something from scratch.

Software for parsing raw data quickly: https://git.projectstratos.nl/mvanheijningen/pc_libs/src/branch/master/rust/storage-parser

16 Kolibri Blackbox format

Writing and reading occurs in 256 byte pages, so it makes the most sense to encapsulate blackbox data into 256 bytes.

Header should contain:

- Serial number
- Time
- Packet type

The packet type determines how the rest of the 256 page should be treated and parsed. The default packet just contains basically all the data. What follows are measurements and system states which are useful to include

- State
- Raw accel

- Raw gyro
- Raw baro
- Computed altitude
- Computed vertical velocity
- Battery voltage
- Continuity detection
- Servo angle
- Pyro status
- GPS coordinates
- Temperature
- Pin states
- Used blackbox capacity

17 Kolibri Telemetry Format

Proposal for a standardised over the air telemetry packet format for the 868 MHz band. Inspiration is derived from the CCSDS space telemetry format and the AltOS telemetry packets used in Altus Metrum products[24]

Telemetry data packets are fixed length at either 16 bytes or 32 bytes. Note that for a given mission, you are limited to using either 16 bits or 32 bits packets, since this needs to be fixed on both the ground station and transmitter side. The packet formats are referred to as: [packetsize]-[Hex ID]. E.g. Packet type 32-01 refers to the 32 byte packets with packet ID of 0x01.

Note that error detection/correction will be handled separately (by the software layer or transceiver).

Endian-ness of the packets are not specified, since they can be arbitrarily defined, but in general, values spanning multiple bytes are to be encoded in a Big-endian manner. For instance, 0xA2B4 (41652 as unsigned integer) should be encoded with 0xA2 in byte 0, and 0xB4 in byte 1.

Package header (4 bytes):

- uint8: Device type
- uint8: Device serial number
- uint16: device time in 100ths of a second (loops around every 10 min)
- uint8: packet type. I.e. how rest of data should be treated

17.1 16 byte packets

These are compact packets used for high rate telemetry. this is useful if you for instance are only interested in the altitude information. Excluding the header, 12 bytes are available for data.

Type 0x00: Debug package. Each byte is simple the byte number:

- Byte 0: 0x00
- Byte 1: 0x01
- Byte 2: 0x02
- ...
- Byte 11: 0x0B

Type 0x01: Dynamic high resolution (16 bit) motion sensor data

- Gyro: 48 bits
- Accel: 48 bits

[24] https://altusmetrum.org/AltOS/doc/telemetry.html

12 bytes

Type 0x02: Altitude/pressure data

- Raw pressure: 24 bits
- Reference (ground) pressure: 24 bits
- Temperature: 16 bits
- Height: 16 bits

Type 0xXX: Ping request. This is a special packet that signals that the transceiver pings a ground station and awaits a response. After transmitting this package, the transceiver switches to receive mode for 100 ms, and the targeted ground station may optionally transmit a telemetry packet using the same packet format.

- Byte 0: Ground station serial number
- Byte 1-11: Reserved

Type 0xXX: Kolibri uplink command. This packet is designed for uplink purposes. E.g. this could contain signals to turn on a camera or remotely deploy a parachute. Three commands can be sent at one, and the command(s) is only executed if all three redundant messages match. Note that this only works if the flight computer is actively listening (e.g. after a ping request or in idle mode). Note: Using uplink will block the reception of any downlink messages (even if using two ground stations).

- Byte 0: Flight computer serial number
- Byte 1-3: Command 1 (3x redundancy)
- Byte 4-6: Command 2 (3x redundancy)
- Byte 7-9: Command 3 (3x redundancy).
- Byte 10-11: Reserved

The 8-bit commands are:

- 0x00: Do nothing
- 0x01: Enable buzzer
- 0x02: Disable buzzer

Type 0xXX: Signed/secure message. A message with a digital signature to ensure authenticacy.

- Byte 0: Target serial number
- Byte 1: Random nonce
- Byte 2-10: Reserved
- Byte 10-11: Signature

This might still be susceptible to replay attacks, but it's better than nothing...

17.2 32 byte packets

These have 28 bits for the data

Type 0x00. Debug package. Each byte is simply the byte number:

- Byte 0: 0x00
- Byte 1: 0x01
- Byte 2: 0x02
- ...
- Byte 11: 0x0B

Type 0x01: Kolibri detailed telemetry packet v1. Default packets containing everything you need

- Flight state: 5 bits
- Pin states: 6 bits
- Vbat: 8 bits
- Baro: 16 bits
- Accel: 24 bits
- Gyro: 24 bits
- Temperature: 8 bits
- More...

Type 0xXX: ASCII text. The data should be interpreted as ASCII text.

- Byte 0-27: Bytestream for ASCII

Type 0xXX: Raw LBP stream. This encapsulates the LBP stream as defined in the Data link layer. E.g. 0x55 is the frame start marker and 0x5A is the frame end marker. Consult the launchbox protocol documentation for more information. Unused bytes (e.g. outside LBP data frames) should be set to 0x00.

- Byte 0-27: LBP bytestream

Type 0xXX (Maybe?): Start LBP packet over the air. This packet encapsulates the launchbox protocol packet format. For more information, consult the LBP documentation. The very last byte in the payload stream contains the CRC of the whole LBP packet.

Each time this packet is received, a new LBP packet will be "built" until the payload length has been reached. Which may require additional packet transmissions of type 0xF1.

- Byte 0: Payload length
- Byte 1: LBP Header 1
- Byte 2: LBP Header 2
- Byte 3: LBP Header 3, the Message ID
- Byte 4-27: LBP payload+CRC. Max 24 bytes

Type 0xF2 (Maybe?): Continue LBP packet over the air.

- Byte 0-27: LBP payload+CRC. Max 28 bytes

18 Duplex and LBP-over-air

So... proper two way communication would be nice, especially if people want to adapt the packet telemetry for something like the launchbox protocol. This can be implemented with time-division duplexing (half-duplex), where the transceivers switch between the transmit and receive role. The tricky part is to figure out when and how they should switch, so here are just some considerations.

One way would be to have a master and slave device. The master initiates communication, and may request a response from the slave device. The advantage here is simplicity, but the the communication link becomes asymmetrical.

The other would be to have synchronized clocks between the two systems, with allocated time slots for transmission and reception. The tricky part here is the synchronization, which would be based on the packet reception time.

The third option would be to use a low enough duty cycle that the chance of packet collision is minimised, and maybe use ack messages to confirm reception. This would be the easiest

implementation, but has limited data rate. This method could take inspiration from LoRaWAN and other WAN implementations.

The "easiest" option would be to have two transceivers in each device, operating at different frequencies.

LBP supports routing and other stuff, which would be difficult for a peer-to-peer network. For this reason, let's focus on implementing a subset of the LBP protocol for communicating between two devices, A and B. The goal is to find a way of making them work as a bridge for LBP serial (i.e. plug one serial cable into module A and one into module B, and have it *just work*̂tm). It should also be resistant to interference/tampering since the commands may be critical (encryption???)

The wired LBP link is serial at 38400 baud, with 8 data bits, no parity and 1 stop bit.

Inspiration:

- https://ieeexplore.ieee.org/document/7249318
- https://www.sciencedirect.com/topics/engineering/time-division-duplexing

Suppose transmitting fixed 32 byte packets, and suppose air data rate is 50 kbps. Each packet then takes roughly 6 milliseconds. Let's round up to 10 ms per packet. Let's say that we're running 25 Hz packet rate for both tx and rx. Meaning each device transmits every 40 ms In other words, the synced communication format would look something like:

```
A Tx:OOOOOOOOOO------------------------------OOOOOOOOOO--
B Tx:--------------------OOOOOOOOOO----------------------
ms  :0---------10--------20--------30--------40--------50
```

When reading or writing data to the serial interface, they should be self-contained LBP packets. Smaller packets can fit within a telemetry packet, but larger ones may need to be split up...

The symmetrical two-way communication can be implemented by having the following logic on both modules.

```
boot()
# check for active devices to sync with
reception_time, rx_data = receive_packet(timeout=5000ms)

while true:
    sleep_until(reception_time+20ms)
    tx_start_time = send_packet()
    reception_time, rx_data = receive_packet(timeout_until=tx_start_time+40ms)

send_packet():
    if tx_data:
        send(tx_data)
    else if long_time_no_send():
        send(tick)
```

Assumin the device clocks are fairly accurate (which they are), this the two loops should lock, allowing for duplex communication at 25 Hz. This approach can be modified to allow faster rates using variable transmit times, by detecting channel activity and transmitting earlier if no channel activity is present, but that's more work and one has to be careful not to mess up the sync.

It won't really help for half-duplex communication, but it could be an idea to define two frequencies for communicating in the two directions but meh... probably not needed for now

19 Misc hardware/software ideas

Other stuff that could be nice to have:

- Enclosure with tripod mount for the ground station
- Expansion board with connections for BOOT, SWD, and a picoprobe for easy debugging.
- Ground station could use a small i2c OLED screen to display stuff.

Since the pico is dual core, it would be possible to run a system in the loop simulation with the simulation running on the second core.

20 Ground station design

The ground station will use one kolibri board, and can either be connected to a laptop or used as a standalone device.

- One button
- OLED screen
- USB port
- Flash chip

32 byte telemetry packets are stored as 64 bytes, in order to include additional information about the receiver (e.g. receiver GPS coordinates, rx sensitivity etc)

The one button handles user interaction. In particular the following:

- Switch between different display screens
- Delete telemetry logs

When turned on normally, the ground station will go into ground station mode, where it continuously receives data and shows it on the screen and over USB. The screens are: **Default telemetry**

- Signal
- Altitude
- GPS coordinates
- Log usage percent

Recovery mode

- Shows distance to last GPS packet and heading arrow.
- Show GPS coordinates and accuracy
- Show time since last packet.

The startup sequence is as follows:

1. 5 seconds of startup/version "Kolibri ground station v1.0", ("Press button to delete logs")
2. Go into default telemetry mode

21 Logbook

The content of the document so far has not been chronological, but instead grouped by topic, with quite a few revisions as things change. This section serves as a chronological-ish timeline.

Before 30/10/2022: Looked into pyro actuation circuitry and had some weird ideas...

30/10/2022: Started procrastinating on exam prep by doing electronics stuff. AKA how this project started. Brainstormed top level goals (super cheap and super compact). Started by trying things in EasyEDA (with one-click JLCPCB assembly) for initial cost estimates.

04/11/2022: Started proper project in Altium (though KiCad was considered). Created new schematic symbols and footprints with JLCPCB part numbers.

04/11/2022 - 11/11/2022: Write up most of the requirements, first iteration schematic design. Wrote "user manual" to guide project development. Component selection and extensive cost optimization. Rough PCB blocking and planning. Wrote down initial ideas for telemetry packet design and software scheduling.

10/11/2022: Finished with exams. Can now feel less bad about wasting time on this.

11/11/2022: Started a bit of PCB routing (it's going to be difficult to fit everything within the space contraints...), fixed some stuff in the schematic. Also stole some Pi Picos + LoRa modules from Leo for testing various stuff. Set up C/C++ SDK for the RP2040 and added helper scripts to build and program. Got basic blink and USB serial communication working. Wrote serial viewer based on pyserial and tested bootloader activation over USB. Started this logbook.

12/11/2022: Prepared some stuff for testing a prototype of the actuation circuit (power supply, components etc). Debug some C++ compiler weirdness and add darestl.

13/11/2022: Looked into DARE-MCS integration. A bit more work on the schematic. Got basic SPI code working. Continue building sketchy lab power supply. Plan out external connectivity interface.

15/11/2022: Finished power supply, promising initial tests of current limiting circuit based om LM317.

18/11/2022: Experimenting with PIO on the pico dev board

24/11/2022: Set up git version control and write build instructions. Set up doxygen and unit testing. Implement basic flight state machine and unit test for it.

25/11/2022: Implement basic 1D launch simulation and investigate dual core in-system simulation. Implement a basic non-preemptive task scheduler and tested multitasking with state machine.

26/11/2022: Clean up some firmware. More studying about task scheduling. Debug and test scheduler on pico hardware.

27/11/2022: Test data storage in flash memory. Plan out procedure for erase/write/read of persistent storage.

30/11/2022: Apparently the cheap GPS module is now out of stock, investigate alternative options. Let's just exclude it for now.

01/12/2022: Start framework for non-volatile data handling.

02/12/2022: Build and document pyro test setup

03/12/2022: Pyro circuit testing and data analysis/visualization

04/12/2022: Finish documenting pyro testing. Tweak schematic and more work on PCB design

05/12/2022: More work on schematic and PCB layout

06/12/2022: Finalize schematic and expansion pinout. Solve all reported schematic errors and fixed some other errors.

07/12/2022: Fix stuff from schematic review. More work on PCB design. Fixed clearance problems and updated pinheader connections. Announce project to DARE.

08/12/2022: Looked into alternative connector options after feedback. Looked into better quality sensors and switched to using the BMI270 and BMP388.

9/12/2022: Spend a few hours on PCB design. Changed out the power supply topology to support 1s lipo (on 5V rail) and move more components to the top layer. Switched to a more

available transceiver board. Create and check footprints for BMI270, BMP388, transceiver. Laid out new power supply components. Create combined power/arming connector.

10/12/2022: Not much, a little work on the PCB design.

11/12/2022: Look into more GPS options, a bit of PCB routing

14/12/2022: A bit more PCB routing for the sensors. Look into antenna options

28/12/2022: Route remaining low speed signals after a bunch of procrastination. Old GPS now back in stock.

30/12/2022: Set up notion database. Initial round of PCB tweaks. Put design files up for review.

31/12/2022: Tweak footprint orientations for pick/place assembly. Tweak silkscreen. Add SMT assembly tooling holes

02/12/2022: Change to M3 mounting holes, other minor tweaks.

04/01/2023: PCB Review and fixes

06/01/2023: Final review and ordering of initial batch of five boards (v1.0) + GPS breakouts.

07/01/2023: Got DFM review back, RF module wrong orientation due to misplaced silkscreen marking. Increment revision to rev 1.1

19/01/2023: Boards received from JLCPCB and film unboxing video + blink test. Found that high current trace for actuation channel 2 was mistakenly decreased in width :/ Easy fix though

20/01/2023: Check that communication works with Baro, IMU and Transceiver. Test out GPS module with phoneRent

25/01/2023 - 06/02/2023: Work on the drivers and firmware

07/02/2023: Work on telemetry encoding/decoding. Working LoRa transmission/reception

29/03/2023: Wrote basic data logger code for Operation Cuckoo (Flight test on Cansat launcher).

31/03/2023: Kolibri #3 crashed, and blackbox chip was lost. In the future, secure flash chip better (e.g. blob it with some glue).

04/04/2023: Update design slightly, JLCPCB changed cheap vias to 0.3mm hole diameter.

12/05/2023: Flew Successfully on Pyrocket and Longboi. GPS tracking used to find the longboi rocket.