ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and
Electrical Engineering

Spring Semester 2018

Multi-Sensors Control System for a Transportation Vehicle in a Low-Pressure Environment

Bachelor Thesis

Carl Friess
cfriess@student.ethz.ch

26 August 2018

Supervisors: Dr. Michele Magno, michele.magno@iis.ee.ethz.ch

Professor: Prof. Dr. Luca Benini, lbenini@iis.ethz.ch

Acknowledgements

Thank you to the Integrated Systems Laboratory[1] for enabling Swissloop to build both Escher and Mujinga by providing resources, help and know-how. In particular, thank you to Dr. Michele Magno for supervising this Bachelor Thesis and providing invaluable advice.

Furthermore, I would like to thank Swissloop as an association for providing the framework for this Bachelor Thesis and as a team which has worked tirelessly over the past year to design and build a Hyperloop pod with unprecedented performance. I am very grateful to Hanno Kappen for designing the PCB which in combination with the Texas Instruments Launchpad provided the platform for this Thesis. Thank you also to Laurin Paech for developing the Control Panel which is closely tied to the software developed in this Thesis.

Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor. For a detailed version of the declaration of originality, please refer to Appendix C

Carl Friess,
Zurich, 26 August 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Hyperloop Passenger Transportation Concept was initially proposed by Elon Musk in his Hyperloop Alpha paper[3] as an alternative to the planned high-speed rail project connecting San Francisco and Los Angeles. It was argued that the high-speed rail project is not state of the art in terms of technology, it is much too expensive and significantly slower than other high-speed trains around the world. The objective of the Hyperloop is to achieve passenger transport on the ground over long distances at speeds exceeding 1220 km/h.

To achieve such high speeds, pressurized passenger capsules ("pods") would run in tubes where near vacuum is maintained. The initial proposal also called for air-bearing to allow the pod to levitate during transit. In order to supply the air bearings with pressurised air and further reduce drag, a compressor would suck the remaining air in through an inlet at the front of the pod. A linear motor system would be used to accelerate and decelerate the pod at high speeds while limiting the acceleration to 1g for passenger comfort.

1.1. Hyperloop Competition

Although the Hyperloop Alpha proposal was turned down, SpaceX decided in 2015 to hold a student competition[4] in order to drive the development of Hyperloop technology. To this end they constructed a 1,25km test tube designed to reach an ambient pressure of 8mBar. The tube features an aluminium sub-track and rail mounted on a concrete fill bed.

Figure 1.1.: Hyperloop test tube at SpaceX headquarters in Los Angeles.

Since the first competition there have been a second and third iteration and a forth has been announced for the summer of 2019.

The objective for the teams is to build a prototype Hyperloop pod and race it in the test tube. The pod reaching the highest velocity with successful deceleration wins the competition. During the first and second competition, a pusher vehicle was available to accelerate the pods to a pre-defined velocity at the beginning of their run. Therefore, it was optional for a pod to incorporate a propulsion system. However, in the third competition the pods were required to accelerate independently.

The first step of the competition is the Preliminary Design Briefing in which the team must outline the main concepts of their pod design. After it has been approved teams may proceed to submit the Final Design Briefing a few months later. This must include all details of the pod's design and show that the design is safe. Approximately 20 teams are then selected to compete in the competition at SpaceX headquarters in Los Angeles.

The competition in Los Angeles consists mostly of a testing week where the pod must pass a series of tests to prove safety and correct operation before being allowed to enter the test tube. The most promising teams are then selected to compete in the final on

the last day of the competition. Here teams aim to reach the highest speed in order to win the competition.

1.2. Swissloop

Swissloop[5] was founded as an association in September 2016 by a group of ETH Zurich students with the intention of competing in the second iteration of the Hyperloop Pod Competition. Swissloop was able to gain support from many industry sponsors and several departments at ETH Zurich including the Integrated Systems Laboratory. In July 2017 Swissloop revealed it's first pod Escher to the public. After reaching the finals of the competition with Escher in August 2017, a new team was assembled to compete in the third competition in July 2018 with a completely new pod called Mujinga.

1.2.1. Escher

Swissloop's first pod featured a cold gas propulsion system which was designed for a second acceleration stage after the initial acceleration delivered by the pusher. The pod also featured hydraulic bakes as well as a passive levitation system.

The avionics implemented for Escher included around 30 sensors and provided a reliable basis for controlling the pod. Although eventually several flaws became apparent, the system provided a solid basis for the development of the avionics system of Mujinga. While Mujinga retained some components of the hardware, the design ended up being completely different in several ways. The Software was almost entirely rewritten from the ground up leading to large performance and reliability improvements.

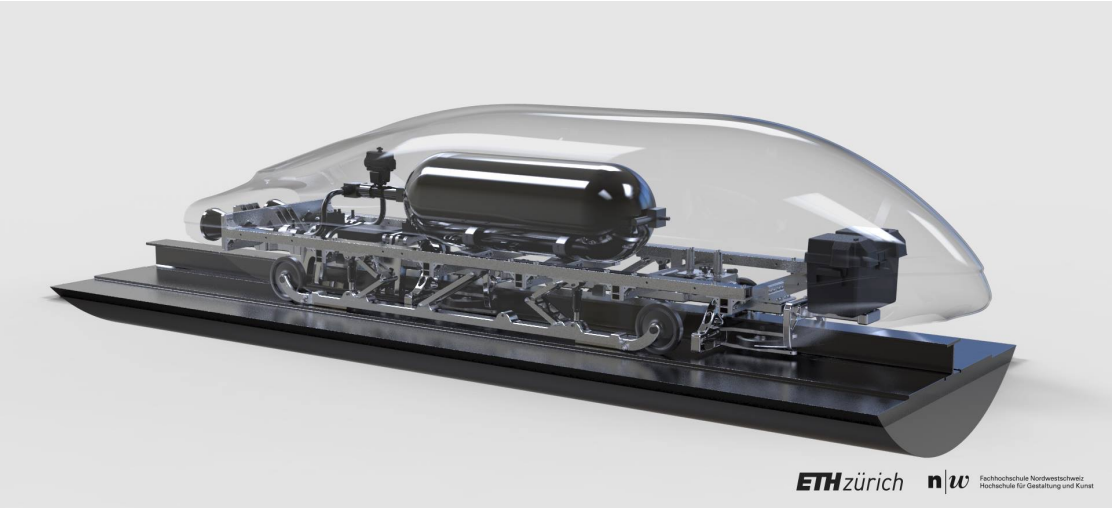


Figure 1.2.: Rendering of Escher with transparent shell.

1.2.2. Mujinga

Swissloop's second pod design builds on the lessons learned with Escher but includes many significant changes. Most noticeable, Mujinga no longer levitates but uses wheels and four electric motors as a propulsion system. Two high-voltage (700V) batteries produce 500kW of power to accelerate to a top-speed of 500km/h. Similar but redesigned hydraulic brakes decelerate the pod before the end of the 1,25km test track. A pneumatic clamping system presses the pod against the track to produce the down-force necessary to achieve the necessary acceleration.

As mentioned, the avionics system was based on the platform used in Escher. However, much emphasis was placed on greater simplicity and reducing bottlenecks. A problem with the logging system in Escher meant that the amount of data that could be acquired was very small. Therefore, the logging system in Mujinga was specifically designed to handle much higher data rates.

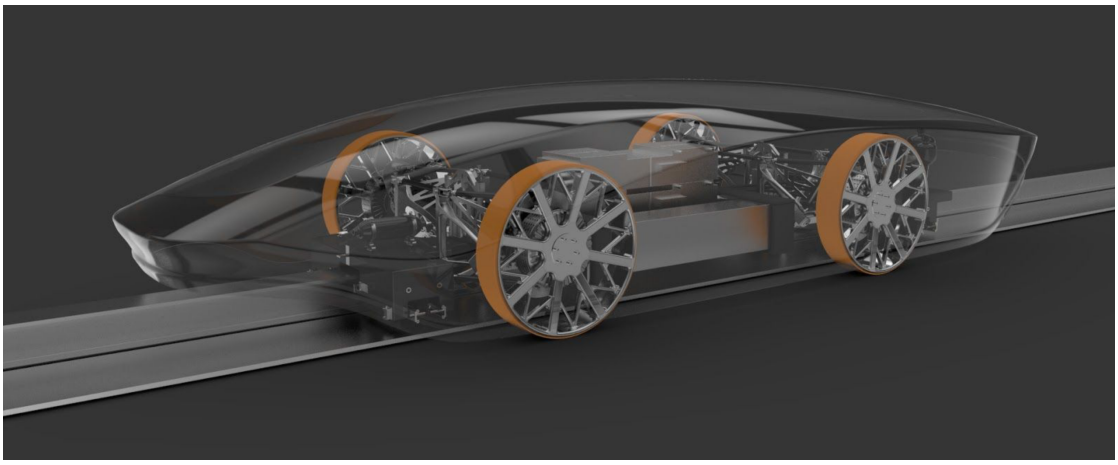


Figure 1.3.: Rendering of Mujinga with transparent shell.

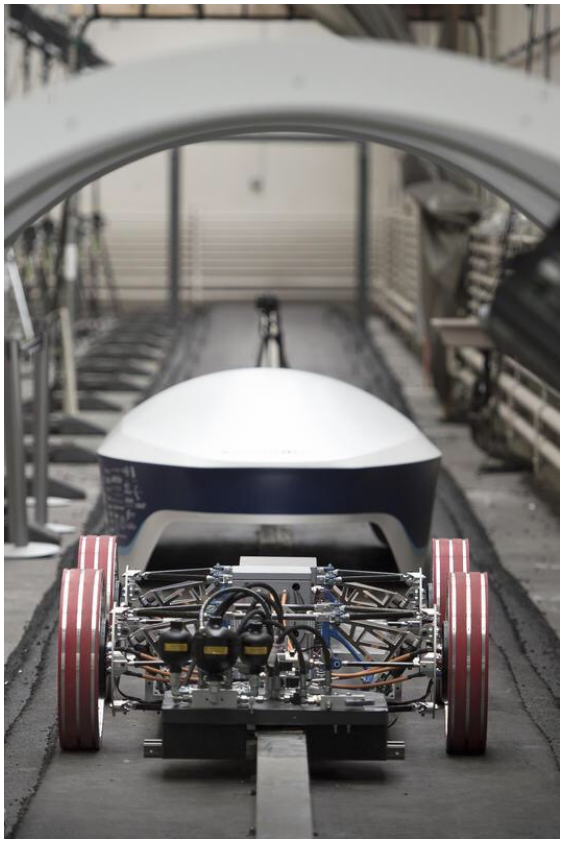


Figure 1.4.: Picture of Mujinga with separated shell.

1.3. Project Scope

The scope of this Bachelor Thesis is the implementation of the avionics and control software running on the Hyperloop pod. This includes the following tasks:

- Platform selection
- Development of drivers to interface and communicate with on-board sensors
- Development of drivers for a network interface and SD card
- Development of drivers for communication with motor controllers (inverters), battery management systems and brake actuators
- Implementation of a control scheme which ensures safe and correct operation of the pod while executing traction control and yaw control algorithms (developed by team members)

- Implementing communication with a control panel (developed by another team member) over a network and logging all collected data, as well as system events

The following tasks are not part of this bachelor thesis and were completed by other people:

- Design of custom PCBs (Hanno Kappen)
- Development of traction control (Julius Wanner and Stefan Weber) and yaw control algorithms (Yannick Strümpler)
- Development of a control panel for visualizing telemetry and controlling the pod (Laurin Paech)
- Wireless network for communication with the pod (SpaceX)

The control system in this thesis was developed specifically for the Hyperloop Pod Student Competition. As such it was tested in the field as part of the Mujinga pod at testing facilities in Switzerland and at the competition in Los Angeles at SpaceX's headquarters.

Chapter 2

System Overview

2.1. Sensors

2.1.1. Laser Distance Sensors

In total four laser distance sensors were used to assess the vehicle attitude in relation to the rail.

Two high precision sensors were employed to measure the lateral alignment to the track at the front and the back of the pod. These sensors provided the input for the yaw controller in order to actively ensure that the pod is correctly aligned with the track and not exercising an torque on the rail.

Two smaller form-factor sensors were also installed at the front and back of the pod to measure the pods vertical alignment. These were mostly used to assess the performance of the clamping system.

2.1.2. Pressure Sensors

Ambient pressure sensors

A high-precision pressure sensor was installed to monitor the ambient pressure. Two further ambient pressure sensors were installed inside of each high-voltage battery pack, as these were pressurized. If the pressure inside the battery packs drops too low the battery cells could be permanently damaged. Therefore, it was necessary to monitor these values and re-pressurize the pod's environment in the event of a leak.

Braking pressure sensors

Four high-pressure sensors were installed in the braking system. One in each braking piston to measure the pressure with which the brakes actuate and to determine their status. Additionally, one sensor was installed in each reservoir holding the pressure used to engage the brakes in order to monitor brake health. The braking system consisted of two independent hydraulic systems for redundancy, thus two sets of sensors were necessary.

2.1.3. Navigational Sensors

Two laser contrast sensors were used to detect optical marking on the wall of the test tube. The optical markings occur in intervals of 30m and can therefore be used to determine the location of the pod along the tube and calculate it's velocity.

2.2. Propulsion system

Four two-phase electric motors are used to accelerate the pod. Each is driven by a separate inverter. The inverters are controlled via a CAN bus and two digital safety signals. The RFE signal enables the inverter and the RUN signal connects the high voltage from the battery to the motor. The inverter can be configured over the CAN bus. Subsequently, both torque and speed commands can be given to the inverters to drive the motors. Furthermore, the inverters provide telemetry over CAN including the following:

- Inverter status
- DC bus voltage
- DC current
- Motor RPMs
- Motor RMS current
- Inverter temperature
- Motor temperature

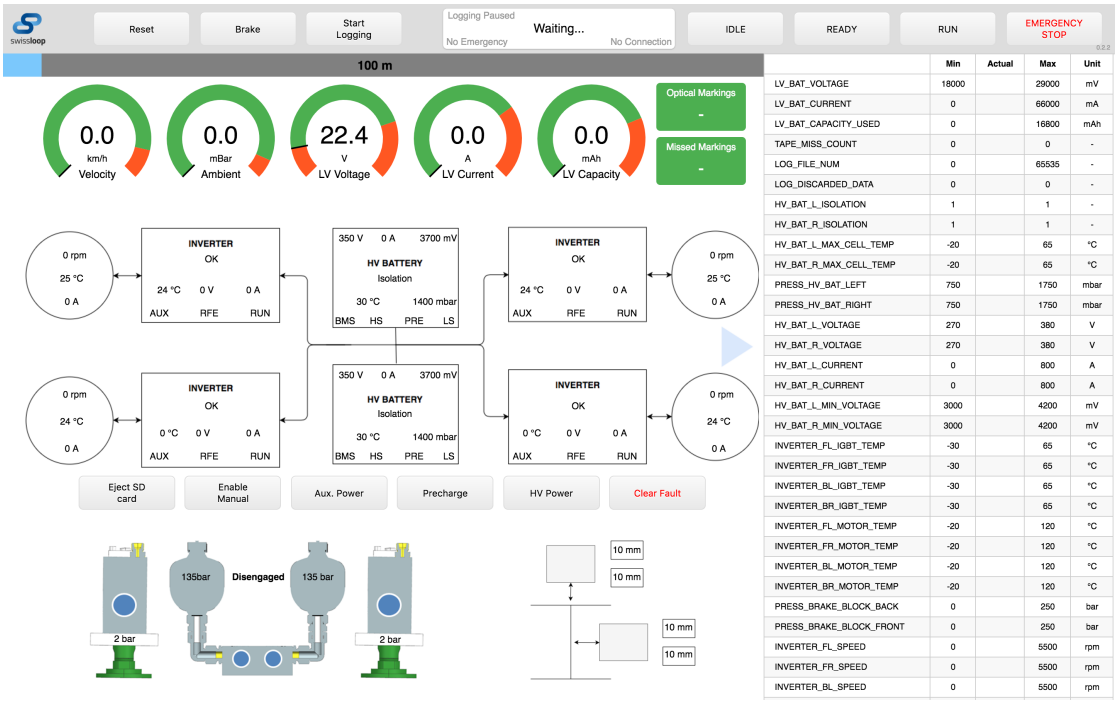
2.3. Battery Management System (BMS)

Two battery management modules (one in each battery pack) are responsible for balancing the battery cells and monitoring them. These modules are also connected to a CAN bus and provide the following telemetry over it:

- High-voltage isolation status
- Battery Pack Voltage
- Discharge/Charge current
- Lowest cell voltage
- Highest cell temperature

2.4. Telemetry and Control Panel

To monitor the pod a network is made available inside the tube. The pod connects to this network and must transmit all telemetry necessary to assess the pod's state and make sure it is safe. In addition, the pod is controlled over the network. Should the connection to the pod fail at any point, the pod must enter a safe state immediately. To control the pod and display telemetry a control panel application was developed.



	Min	Actual	Max	Unit
LV_BAT_VOLTAGE	18000		29000	mV
LV_BAT_CURRENT	0		66000	mA
LV_BAT_CAPACITY_USED	0		16800	mAh
TAPE_MISS_COUNT	0		0	-
LOG_FILE_NUM	0		65535	-
LOG_DISCARDED_DATA	0		0	-
HV_BAT_L_ISOLATION	1		1	-
HV_BAT_R_ISOLATION	1		1	-
HV_BAT_L_MAX_CELL_TEMP	-20		65	°C
HV_BAT_R_MAX_CELL_TEMP	-20		65	°C
PRESS_HV_BAT_LEFT	750		1750	mbar
PRESS_HV_BAT_RIGHT	750		1750	mbar
HV_BAT_L_VOLTAGE	270		380	V
HV_BAT_R_VOLTAGE	270		380	V
HV_BAT_L_CURRENT	0		800	A
HV_BAT_R_CURRENT	0		800	A
HV_BAT_L_MIN_VOLTAGE	3000		4200	mV
HV_BAT_R_MIN_VOLTAGE	3000		4200	mV
INVERTER_FL_IGBT_TEMP	-30		65	°C
INVERTER_FR_IGBT_TEMP	-30		65	°C
INVERTER_BL_IGBT_TEMP	-30		65	°C
INVERTER_BR_IGBT_TEMP	-30		65	°C
INVERTER_FL_MOTOR_TEMP	-20		120	°C
INVERTER_FR_MOTOR_TEMP	-20		120	°C
INVERTER_BL_MOTOR_TEMP	-20		120	°C
INVERTER_BR_MOTOR_TEMP	-20		120	°C
PRESS_BRAKE_BLOCK_BACK	0		250	bar
PRESS_BRAKE_BLOCK_FRONT	0		250	bar
INVERTER_FL_SPEED	0		5500	rpm
INVERTER_FR_SPEED	0		5500	rpm
INVERTER_BL_SPEED	0		5500	rpm

Figure 2.1.: Screen shot of the control panel.

2.5. Data Logging

In order to verify the pod's performance and diagnose possible failures, we wanted to be able to log as much data as possible. Therefore, we set a goal of logging all sensor data that comes into the system.

2.6. Control and State Machine

Overall, the pod is controlled by a finite state machine (FSM). The FSM must incorporate all critical safety checks and is also used to allow the pod to complete the run autonomously. In order to improve testability, the FSM should have the minimum number of states necessary for the desired functionality. To this end we also decided to minimize the number of automatic state transitions, minimizing the risk of bugs in the implementation.

2.7. Correctness and Testability

The highest priority for this system was to ensure correctness and safety. Therefore it was important to minimize sources of errors and implement the system with testability in mind. Tests include unit tests for individual control sequences and algorithms, as well as functional tests.

2.8. Platform

The hardware platform used for this project is a combination of a custom designed PCB and a Texas Instruments Launchpad (LAUNCHXL-F28379D)[6] featuring the TMS320F28379D micro-controller[7]. The Launchpad provides a sold basis which includes all the components necessary to run the micro-controller. It then plugs into the custom PCB which accommodates all the necessary external components and incorporates connectors for all sensors and actuators.

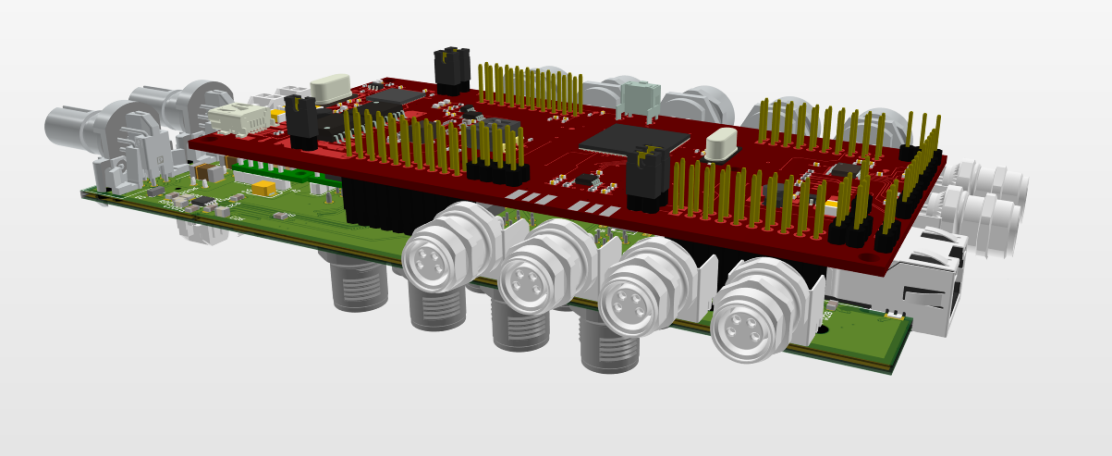


Figure 2.2.: Custom PCB with Texas Instruments Launchpad.

2.8.1. Micro-controller

The Texas Instruments TMS320F28379D was chosen as it provides high performance in terms of processing with two C28x CPU cores running at 200 MHz and two Control Law Accelerators (CLA). In addition, it incorporates a wide range of versatile peripherals covering most types of interfaces used in the system. Furthermore, this processor is a dual-core version of the single core TMS320F28377S which proved to work very well in Escher.

Texas Instruments also provides a good Runtime Support Library (RTS) for C, as well as header files and support functions as part of ControlSUITE[8] making it easier to access device registers.

2.8.2. Ethernet Controller

To establish a network connection on the test track an Ethernet connection was required. Since the micro-controller is not equipped with an Ethernet interface it needed to be included externally.

After considering several options we decided on the WIZnet W5500 Ethernet Controller. Beyond providing Ethernet support it also incorporates hardware implementations of ICMP, ARP, IPv4, TCP, UDP and other protocols. This is advantageous as it offloads the computation necessary to run the network stack from the main processor, providing better performance. Furthermore, the chip is widely used and therefore has good community support.

The micro-controller communicates with the Ethernet Controller over SPI but the W5500 also provides an interrupt line which can be configured to provide interrupts on events such as incoming packages.

2.8.3. SD-Card

In order to log telemetry data a form of non-volatile memory was needed. The data must be easily accessible and quickly retrievable. The obvious and most suitable choice is an SD-card, as it can be integrated into the SPI bus and provides large amounts of storage. At the same time it can be easily plugged into a laptop in order to retrieve the data in the field.

2.8.4. External Analog-To-Digital Converter (ADC)

The pressure sensors on the pod produce analog signals that need to be converted. Additionally, the pod incorporates a set of low-voltage batteries and it is necessary to monitor their voltage and discharge current. Although the micro-controller features a built-in 12-bit ADC which could accomplish this task, we wanted to achieve higher precision using an external 24-bit ADC (ADS124S08). Communication with the ADC also runs over an SPI bus. However, the external ADC uses a different SPI mode than the Ethernet Controller and SD-card. Thus a separate SPI bus is required.

2.8.5. RS485 Bus

An objective in the design of this system, was to use as many digital sensors as possible. This was possible for both types of laser distance sensors used on the pod. Both support the RS485 serial bus. Using an appropriate RS485 transceiver, the Serial Communication Interface (SCI) of the micro-controller can be utilized almost natively to communicate with multiple sensors on a single RS485 bus. Unfortunately the two types of sensors use different bus settings and thus it was simpler to separate them into two buses with two sensors each.

RS485 utilizes differential signalling to transmit words asynchronously in the same format as the more widely used RS232 protocol. Since RS485 uses a single differential signal, it is only possible for one device to transmit at a time.

Using the RS485 bus standard means less analog signals that are more prone to interference. Furthermore, it also allows for higher precision, as there are no precision losses during to conversions.

2.8.6. CAN Bus

The motor-controllers (inverters) used on the pod are designed for automotive applications, while the Battery Management Systems (BMS) are designed for aerospace applications. Both systems are designed to use the CAN bus for control and telemetry. The micro-controller incorporates a CAN bus and transceiver on the Launchpad making communication with these devices possible without any additional hardware.

The main advantages of the CAN bus in this application are built-in bus arbitration and automatic retransmission. This allows devices on the CAN bus to transmit telemetry data asynchronously without the possibility of data loss.

Chapter 3

Control System Implementation

The software for the micro-controller was implemented using the C Programming Language. I considered using a real-time operating system but decided that in this case it would be simpler not to do so. The software runs a main loop and all modules are fundamentally asynchronous. In testing it was possible to show that this yielded very good performance.

Another major design decision is to use only static memory allocation. Although at times this can be limiting and is less space efficient than dynamic memory allocation, it also leads to higher performance. More importantly, this approach guarantees that there are no memory leaks, which can otherwise become fatal bugs that may be difficult to diagnose.

3.1. Structure

Each CPU core runs independently from the other, except during system initialisation. Since some peripherals need to be initialised in the correct sequence and the main loop should not be entered until the entire system is initialised, inter-processor communication (IPC) flags are used to synchronise both cores during start-up.

Each core has dedicated FLASH memory, as well as dedicated RAM regions. Additional shared RAM can be read by both CPUs but only written to by one CPU per block. At boot CPU1 configures the global shared RAM regions according to the application's needs and then sends a command to the boot loader on CPU2 using IPC registers to start the program for CPU2.

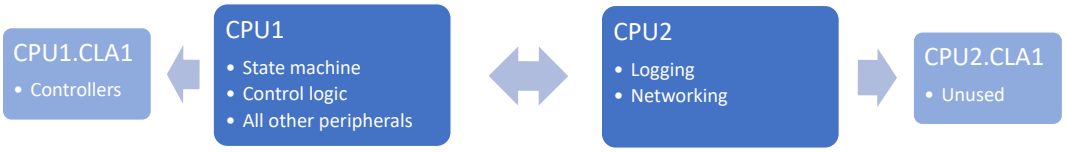


Figure 3.1.: Division of tasks between processors and co-processors.

Figure 3.1 shows the division of tasks between the two CPU cores and one of the Control Law Accelerator (CLA) co-processors. CPU runs most drivers and handles the control logic and state machine. All sensor data is processed on this core and all control decisions are made on it as well. This consolidates the data processing and control logic in one program without the need for concurrency control that could be subject to hard to find bugs.

CPU2 is dedicated to Logging and Networking. The main constraint for this is that the FAT file system is computationally heavy and, as will be discussed in section 3.3, is hard to implement asynchronously. In order to reduce the performance impact, logging was separated onto the second core. However, since the Ethernet controller uses the same SPI bus as the SD-card and only one CPU core can be the master of each peripheral (in this case one of the SPI interfaces), networking also needs to be handled on the second core.

Two control algorithms for traction and yaw control must execute regularly in precise intervals and therefore stand in conflict with the asynchronous nature of the remaining system. Furthermore, as they were developed by other people who did not necessarily have detailed knowledge of the system, it made sense to provide an isolated environment for the execution of the controllers.

The CLA co-processor runs at the same frequency as the associated CPU core but executes code independently. It is assigned areas of the associated CPU's dedicated RAM for program and data space. As the CLA co-processor is designed to run controllers, it can be assigned up to eight function pointers, which represent tasks. These tasks can be triggered from peripherals but also from software running on the CPU. Once a task is triggered, it runs to completion. Although the instruction set for the CLA is separate and significantly more limited, these other features make the CLA perfectly suited for the task.

3.1.1. Module life-cycle

All modules and drivers follow a similar structure. This structure mainly consists of three functions that are called at different times during the module life-cycle as illustrates in Figure 3.2: `init()`, `configure()` and `update()`.

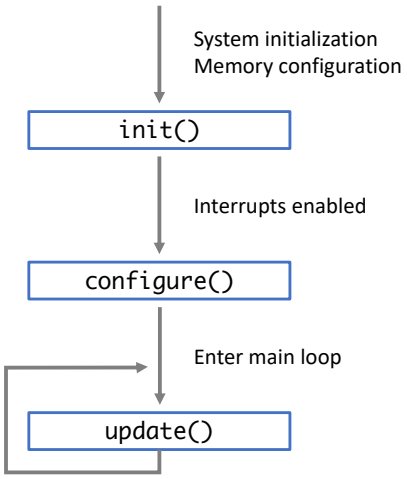


Figure 3.2.: Life-cycle of each module in the system.

Immediately after system initialization and memory configuration is complete, `init()` is called to allow modules to perform initialisation routines, configure peripherals and enable required interrupts. However, interrupts remain globally disabled for the duration of these function's execution.

Some drivers require further initialisation after interrupts have been enabled. Mostly this involves the configuration of an external device such as the Ethernet Controller, SD-card and laser distance sensors. For these initialisation sequences require fully functional communication and therefore interrupts. This is the purpose of the `configure()` functions.

After start-up is complete the processor enters the main loop. At each iteration of the loop the `update()` function of each module is called. If the module does not need to perform any computation at that time it immediately returns. Several modules use internal finite state machines in order to allow for asynchronous execution.

3.1.2. Data-flow

Figure 3.3.: Flow of sensor data and derived data.

All data processing is centralised using the `globals_sensor_data()` function. The concept is to simplify how data is distributed throughout the program. Drivers call `globals_sensor_data()` whenever a new data point is gathered from a sensor. Derived data such as velocity or location is also collected through this function. In this way the data flow is abstracted from the driver layer improving modularisation, maintainability and flexibility.

Once a data point has been collected by a call to `globals_sensor_data()`, the data can be stored to a global structure for use by other modules and used to update the global finite state machine. Additionally, all data points are logged as will be described in Section 3.3.

3.2. Networking

The network protocol needs to be resistant to packet loss and unstable network connections, while still providing a steady stream of data. Unlike the event-based logging system (Section 3.3) telemetry data transmitted over the network can be relatively infrequent and there is no data that must be delivered at a higher rate than other data. Therefore I implemented the protocol using UDP packets. The structure listed in Appendix A holds all the data that needs to be transmitted. It fits into a single UDP packet without segmentation. This makes it very efficient when transmitting telemetry and resistant to occasional packet loss as only one packet is needed to observe the pod's state.

For controlling the pod remotely from the control panel the protocol follows an analogous scheme using a control frame consisting of the following structure.

```
struct ctrl_frame {
    uint16_t set_state;     // 0: D.C. / ..: the state
    uint16_t aux_power;     // 0: D.C. / 1: OFF / 2: ON
```

```
    uint16_t precharge;     // 0: D.C. / 1: OFF / 2: ON
    uint16_t battery;       // 0: D.C. / 1: OFF / 2: ON
    uint16_t brakes;        // 0: D.C. / 1: DISENGAGE / 2: ENGAGE
    uint16_t low_speed;     // 0: D.C. / 1: BACK / 2: STOP / 3: FWD
    uint16_t clear_fault;   // 0: D.C. / 1: CLEAR
    uint16_t reset_run;     // 0: D.C. / 1: RESET
    uint16_t bms_power;     // 0: D.C. / 1: OFF / 2: ON
    uint16_t eject_sd_card; // 0: D.C. / 1: EJECT
};
```

Both the telemetry frame and control frame must be sent (and received) regularly in order for the pod and the control panel to check connectivity and enter a safe state should it fail. Therefore the fields in the control frame are usually zero and a non-zero value indicates a control command.

Although the networking drive primarily executes on CPU2, a small portion also executes on CPU1. This is necessary, since all telemetry data originates from CPU1. Rather than placing data in global structures unnecessarily, the telemetry frame is simply assembled on CPU1 and then read by the DMA controller on CPU2. Similarly, CPU2 receives incoming control packets and CPU1 decodes them and executes any control commands. This process is illustrated by Figure 3.4.

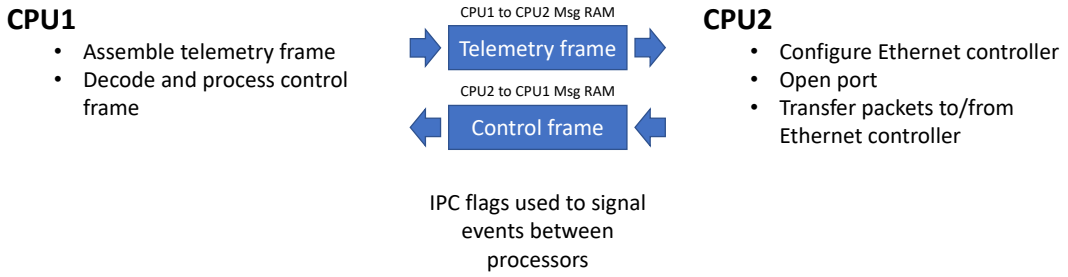


Figure 3.4.: Structure of networking driver.

The telemetry and control frames are written to Message RAM, which is a shared memory area specifically designed for this kind of usage. It consists of two 1K regions, one writeable only by CPU1 and one only writeable by CPU2. The telemetry frame only consists of 76 16-bit words and therefore easily fits into the Message RAM. The microcontrollers Interprocessor Communication (IPC) module further provides flag registers. The networking driver uses IPC flags to notify the other CPU of the following events:

- CPU1 → CPU2: Telemetry frame assembled and ready for transmission.
- CPU2 → CPU1: Telemetry frame has been transmitted and can be overwritten.

- CPU2 → CPU1: Control frame received.
- CPU1 → CPU2: Control frame has been processed and can be overwritten.

To achieve the highest possible performance all SPI communication with the Ethernet controller is implemented using the micro-controller's DMA controller. Since the SPI interface can be used with a 16-level FIFO extension, the DMA controller can operate in Burst mode. This means that the data is transferred in bursts of 16 words until the transfer has completed. The next bust is triggered by the FIFO interrupt which can be configured to assert at a specific level. For this implementation it is configured to trigger when the FIFO is empty but depending on the size of the bursts this can be optimized.

The following listing shows the process of configuring channel one of the DMA controller for transmission of a buffer at `src` of length `len`. Notice that the burst length must be a divisor of the total buffer size. This is because the DMA controller cannot handle bursts of different lengths.

```
void spia_tx_dma(uint16_t *src, uint16_t len)
{
    uint16_t burst;

    // Calculate largest burst length
    for (burst = 16; len % burst; burst /= 2) ;

    // Reset the TX FIFO
    SpiaRegs.SPIFFTX.bit.TXFIFO = 0;

    // Configure DMACH1 for SPIA TX
    DMACH1AddrConfig(&SpiaRegs.SPITXBUF, src);
    DMACH1BurstConfig(burst - 1, 1, 0);              // Burst size, src step, dest step
    DMACH1TransferConfig((len / burst) - 1, 1, 0);  // Number of transfers, src step, dest
    DMACH1ModeConfig(DMA_SPIATX,PERINT_ENABLE,ONESHOT_DISABLE,CONT_DISABLE,
                     SYNC_DISABLE,SYNC_SRC,OVRFLOW_DISABLE,SIXTEEN_BIT,
                     CHINT_END,CHINT_ENABLE);

    // Set the TX FIFO interrupt level
    SpiaRegs.SPIFFTX.bit.TXFFIL = 0;

    // Start the DMA transfer
    _spia_tx_dma_done = 0;
    StartDMACH1();

    // Release the TX FIFO from reset
    SpiaRegs.SPIFFTX.bit.TXFIFO = 1;
```

```
}
```

The procedure for configuring the DMA controller to read data from the SPI interface is analogous.

Using the DMA controller means that the SPI interface can operate at it's highest possible data-rate without causing significant CPU usage. The clock rate used for the SPI bus is 12,5 MHz.

3.3. Logging

3.3.1. File system

The largest design decision in designing the logging driver was the file system to use. The main requirement was for the logged data to be quickly and easily retrievable. The most commonly used file system on SD-cards is FAT (FAT32 or exFAT). The advantage of the FAT file system in this case is that it can be read with almost any computer making it particularly easy to retrieve log files. On the other hand, FAT is very inefficient unless the FAT table (which on larger storage devices is very large) can be cached in memory. If the FAT table is not cached, it must be read block by block every time a file is being written and a new block is needed. In embedded applications like this one, it is usually not possible to cache the FAT table.

Instead of using FAT I also considered employing specialized logging file systems such as log_fs[9] and Yaffs[10]. These sequentially write to the storage medium and do not need to access any other data structures while writing the file. Although this leads to higher performance while writing, it also makes finding and extracting log data much more complex. Furthermore, it is not possible to read the data on a regular computer without specialised software. For this reason, I decided to use the FAT file system and write the log data to CSV files.

Log files are created at system start-up and data is continuously appended until the system shuts down or a manual command is given to close the file and ensure data consistency. The CSV file consists of three columns:

- **timestamp**: The time stamp of the data point measured in milliseconds since system start-up.
- **id**: An integer describing the type of data point or event being logged. IDs are defined by the enumeration listed in Appendix B
- **value**: The value being logged.

The following listing illustrates the structure of a typical log file:

```
timestamp;id;value
4778;41;7
4778;39;348
4778;40;3820
4778;42;24
4778;45;5
4778;43;352
4778;44;3781
4778;46;26
4778;5;0
4778;4;0
4879;38;0
4879;37;0
4879;36;0
4879;1;6
4879;2;4
```

Although beneficial, implementing a specialised FAT driver would not have been possible within the time-frame of this project. Therefore, I decided to use the FatFs[11] library. At first I attempted to use the newest version of the library. However, the micro-controller uses 16-bit addressable memory and therefore does not support 8-bit data types. Without rewriting large portions of the library it was not possible for it to run without native 8-bit data types. Luckily Texas Instruments includes an older version of the library with ControlSUITE[8] that is able to run without 8-bit data types.

The FatFs library provides the file system library but an SD-card driver layer also needed to be implemented to provide the interface to the underlying storage medium through the SPI driver. Unlike all other drivers, the SD-card and FAT drivers are not asynchronous. This is mostly due to how the FatFs library operates. Additionally, the data is organised into 8-bit bytes each contained within a 16-bit word which is the smallest possible data type on the processor. Furthermore, due to this constraint the use of DMA is not practical as data mapping is necessary before data can be written to the SPI interface and after it is read from the SPI interface.

Due to the limitations described above, the entire logging driver is fundamentally synchronous and the SD-card driver is implemented using only the FIFO extension of the SPI interface and without DMA. Given more time, a custom FAT driver could be designed specifically for this processor and incorporate the DMA controller and asynchronous operation. Alternatively, a real-time operation system could be used to partially alleviate the problem using threading. Another crude fix to introduce some asynchronism would be to utilise long jumps to context switch out of the logging driver.

3.3.2. Event-based logging

In the logging system implemented in Escher, logging and telemetry data transmission were coupled. The main disadvantage to this was that all data values were logged at the same frequency and no prioritisation was possible. Although most sensors on Escher were sampled at the same rate, this is not the case on Mujinga where some sensors such as the laser distance sensors are sampled at around 150Hz and others such as the brake pressure sensors are sampled at 10Hz. Thus, in contrast to Escher, Mujinga's logging system is event-based.

Whenever a new data point is acquired or another system event occurs, a string corresponding to a new line in the CSV log file is generated and appended to a buffer. The following listing shows the procedure for generating the string:

```
// Buffers used to construct string
// ( 8) timestamp (ms)
// ( 1) ;
// ( 5) id
// ( 1) ;
// (20) value
// ( 3) \r\n\0
static char string[38], temp[21];

// Write timestamp
itoa(micros() / 1000, string);

// Write ';'
strcat(string, ";");

// Write id
itoa(id, temp);
strcat(string, temp);

// Write ';'
strcat(string, ";");

// Write value
itoa(value, temp);
strcat(string, temp);

// Write "\r\n"
strcat(string, "\r\n");
```

3.3.3. Dual-core structure

Similar to the networking driver, the logging driver is also divided between both cores since all data that needs to be logged is generated on CPU1 while CPU2 maintains the file system and writes to the log file. Figure 3.5 illustrates the cross-core structure.

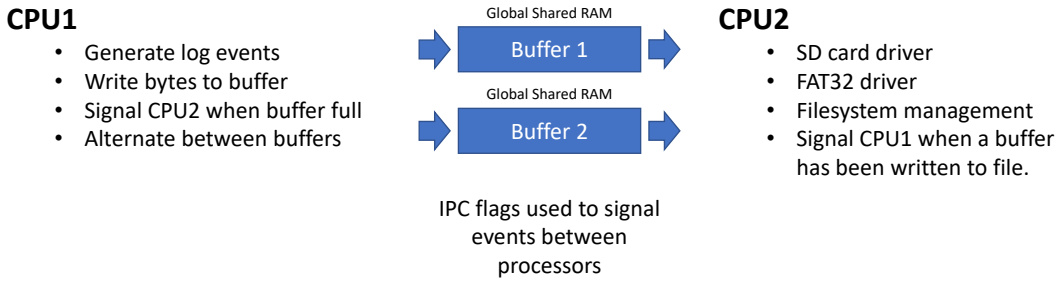


Figure 3.5.: Structure of logging driver.

Two buffers in shared memory (writeable by CPU1) are used to hold the next segments of the log file as they are generated. Once a buffer is full an IPC flag is used to communicate this to CPU2 and CPU1 continues to write to the other buffer. CPU2 writes the buffer out to the SD-card. Once this is complete it uses an IPC flag to signal to CPU1 that the buffer can be overwritten again and the process repeats.

Usually this kind of functionality would be achieved using a ring-buffer. However, in this particular situation this approach does not work as well for two reasons:

Firstly, for performance reasons relating to the SD-card, performance is best when the file is written in consecutive blocks of 512 byte. Additionally, this reduces the overhead produced by reading the FAT table since every block will only be accessed once.

Secondly, to optimise the performance of the FatFs library by reducing overheads, the write operation should be invoked infrequently with large chunks of data rather than frequently with smaller chunks of data. When using a ring-buffer a chunk of data may not be consecutively stored in memory leading to at least two invocations of the write operation rather than one.

Given these constraints, an optimally implemented and configured ring-buffer would operate in an identical way to the separate buffers approach explained above. Each buffer consists of 4kB. The trade-off leading to this size is explained in Section 4.1.1.

3.4. External Analog-To-Digital Converter (ADC)

Some of the code for communicating with the external analog-to-digital converter was based on the software from Escher, since the same component was used. However, the

code has been optimised to reduce communication to 32 SPI clock cycles per measurement.

At start-up the external ADC is reset to make sure it is in a known state. After initialising the ADC, it is configured to use it's internal 2.5V reference and deliver 100 samples per second. The ADC reads data form 10 channels so this sample rate provides a constant sampling rate of 10Hz. Single-shot conversion mode is used so that the ADC provides a single conversion before the channel can be changed for the next conversion.

In the main loop the processor waits for the $\overline{DRDY}$ signal from the ADC to go low indication that the last conversion has completed. Next, the data sequence illustrated by Figure 3.6 delivers the 24-bit value of the conversion and starts the next conversion.

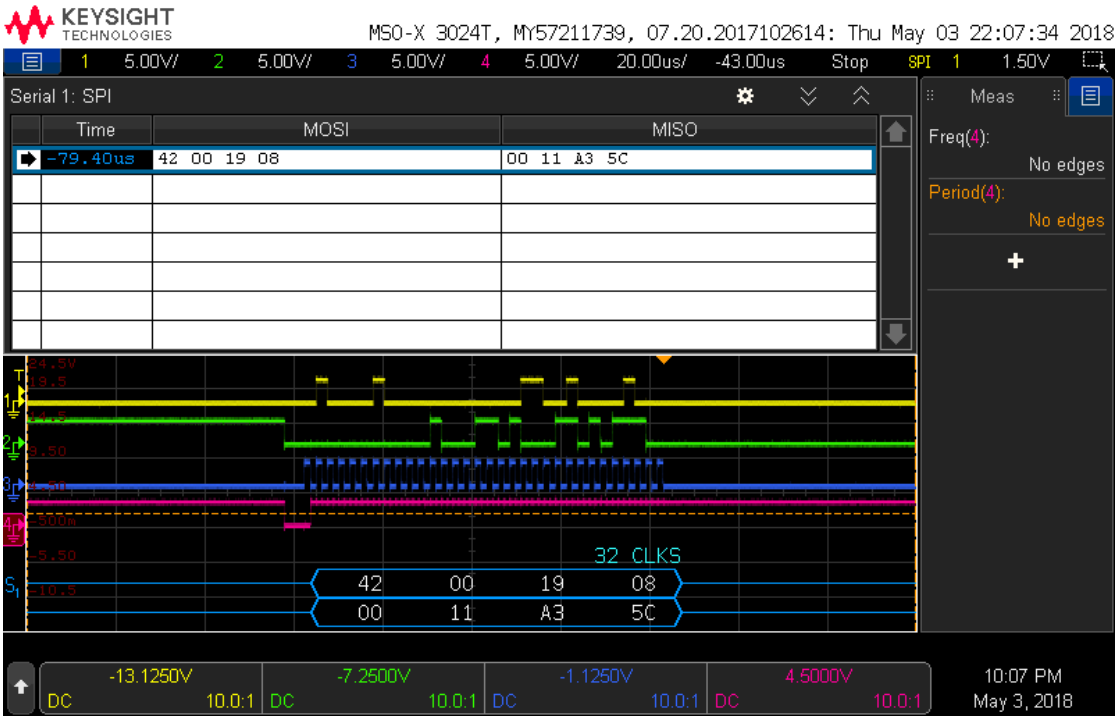


Figure 3.6.: Data flow on the SPI bus with the external ADC.

The following commands and data can be observed in Figure 3.6:

- MCU → ADC: (MOSI)
  - **42 00**: Write one byte to input multiplexer register.
  - **19**: New value for input multiplexer register, selecting next channel.
  - **08**: Start conversion command.
- ADC → MCU: (SOMI)

– **00**: Status byte.

– **11 A3 5C**: Value of previous conversion.

3.5. RS485 Bus

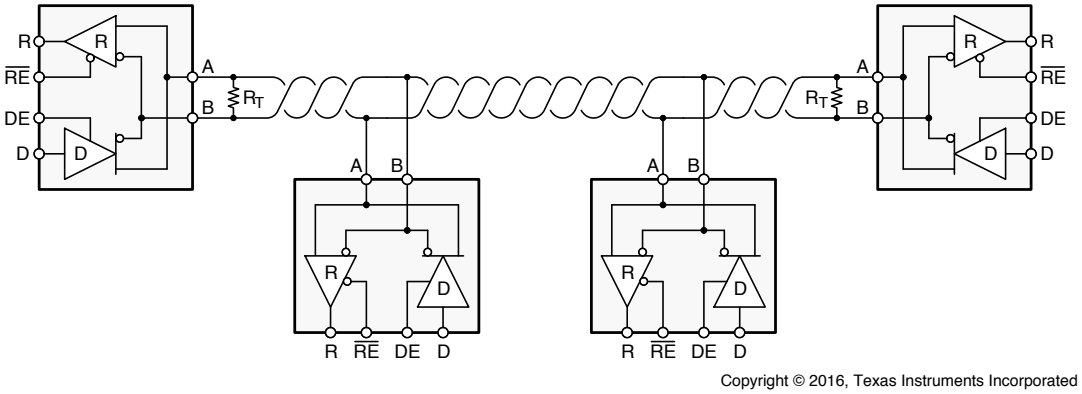


Figure 3.7.: Structure of RS485 bus - Texas Instruments SN65HVD7x Datasheet[2]

As mentioned before, the data format used in the RS485 bus standard is identical to standard serial (RS232) and therefore supported by the serial communication interface (SCI) built into the micro-controller. However, RS485 uses differential signalling. Therefore, a transceiver is required. Additionally, the bus structure means that only a single device may transmit at a time. For the micro-controller this has more complicated implications. The transceiver can only receive data if it is not transmitting, meaning that the micro-controller must produce an RTS signal ($\overline{RE}$ and DE) to determine which state the transceiver is in.

Using the 16-level FIFO extension of the SCI interface, the drivers are able to operate asynchronously by pushing up to 16 words into the FIFO and continue processing while these are transmitted. The need to control the RTS signal would now break this asynchronism since it must be asserted immediately before beginning the transmission and immediately after the transmission completes to allow other devices to use the bus. Next, I will describe a solution which allows for asynchronism again using interrupts.

First, the FIFO interrupt is configured to trigger once the FIFO is empty. However, the interrupt triggers when the last word in the FIFO has been transferred to the transmit buffer. There is no interrupt that can trigger when the last word has been fully transmitted. Therefore, the interrupt service routine triggers a CPU timer which triggers another interrupt after the duration it takes to transmit the final word. This time can be calculated using the BAUD rate. The interrupt service routine for the timer interrupt de-asserts the RTS signal.

3.5.1. OADM

The RS485 protocol used by the Baumer OADM laser distance sensors is simple. Each sensor has a unique ID (a single-digit non-zero number), which is configured during installation. The micro-controller requests the current sensor reading from a specific sensor using its ID and the sensor responds with the reading. The request consists of four characters where `*` represents the ID of the desired sensor: `{*M}`. An example reply from the sensor looks as follows: `{*MM0448564}`.

The following listing shows the `update()` function for the OADM driver module. A simple state machine with two states allows the the module to be reset to the `READY` state, which is also it's initial state.

```
void oadm_update()
{

    uint16_t BUF[12];
    uint32_t reading;
    static uint64_t time_requested = 0;

    switch (state) {

    case READING:
        // Check if we have received the full reply from the sensor
        if (micros() - time_requested > 100000ull) {
            state = READY;
            return;
        } else if (scic_available() < 12) {
            return;
        }

        // Read the sensor's reply
        scic_rx(BUF, 12);

        // Parse the response
        BUF[9] = '\0';
        reading = map(atoi((char *) BUF + 4), 0, 8191, 0, 120000);  // um

        // Emit the reading
        globals_sensor_data(current_sensor == 1 ? SENSOR_DIST_RAIL_TOP_FRONT : SENSOR_DIST

        // Read from the next sensor
        current_sensor = (current_sensor % NUM_SENSORS) + 1;
```

```
        // Fall through to READY case
        /* no break */

    case READY:
        // Prepare the request
        BUF[0] = '{';
        BUF[1] = 'O' + current_sensor;
        BUF[2] = 'M';
        BUF[3] = '}';

        // Send the request
        scic_tx(BUF, 4);

        // Keep track of time
        time_requested = micros();

        // Go to READING state
        state = READING;

        break;

    }

}
```

3.5.2. OM70

The Baumer OM70 sensors use different RS485 bus settings and also employ a more complex protocol. Nonetheless, the principal of operation is the same except that the sensors must be put into an RS485 enabled mode using an RS485 lock command at start-up. However, a problem arises from the reply from the sensor being significantly longer. The following few lines are example responses from the sensor.

```
:01A;108.648;0;2DBB\r\n
:01A;108.577;13;921D\r\n
:01A;NaN;12;B54E\r\n
```

The replies from this sensor are up to 22 characters long. That is more than the 16 characters that fit into the FIFO of the SCI. This is an issue as without using the FIFO the driver cannot operate asynchronously. Using DMA would alleviate this problem but unfortunately the DMA controller on this micro-controller cannot access the SCI.

Furthermore, the length of the reply from the sensor is variable and so the DMA controller would not be well suited to the task.

To prevent a FIFO overflow, an interrupt on the receiving FIFO is configured to trigger when the first 13 characters of a reply have been received. The interrupt service routine then reads the first segment of the reply. The second segment can then be read later without the FIFO overflowing.

3.6. CAN Bus

The CAN bus is designed to be deployed in automotive applications. Mujinga incorporates multiple devices that use the CAN bus. The micro-controller conveniently incorporates a CAN interface and the Launchpad includes a CAN transceiver, so no additional hardware was necessary to support the CAN bus. The built-in CAN interface uses a dedicated Message RAM to store message objects that can be configured to accept CAN messages with an optional CAN ID filter or to hold messages that should be transmitted on the CAN bus. The CAN interface on the micro-controller handles bus arbitration and automatic retransmission so the application only needs to configure message objects with the required properties. To further simplify development using the CAN interface I used a driver library provided by Texas Instruments as part of ControlSUITE[8].

3.6.1. Inverters

The motor-controllers (inverters) only use two CAN IDs each: one to receive and one to transmit CAN messages. The details of what message is being transmitted is contained within the message. This is somewhat in contradiction to the original CAN specification where the CAN ID should identify the type of message as it is used as the priority during bus arbitration.

The driver for the inverters configures two message objects for each inverter configured to accept only the corresponding message IDs. The driver then uses the new data register of the CAN interface to check if a new message has been received in the receiving message objects. To send a message to the inverters, it is simply written into the appropriate transmitting message object and configured to be retransmitted until the inverter acknowledges the message.

To kick-start communication, the control panel sends a "Clear inverter faults" command. This leads to the driver transmitting a sequence of messages to each inverter. The messages complete two tasks: First, clear any faults generated by previous error conditions. Secondly, configure the inverter to regularly send the telemetry listed in Section 2.2. Most measurements are transmitted every 10 milliseconds.

3.6.2. Battery Management Systems (BMS)

Unlike the inverters the Battery Management Systems adhere to the CANopen standard. The BMSs are more passive than the inverters and do not require configuration. They simply start transmitting telemetry data at start-up. CANopen messages can be identified by their CAN ID but still contain additional information in the message.

The driver for the BMSs configures one message object for each BMS and for each message that needs to be received. Each message object is configured to only accept the appropriate message ID. Similarly to the inverter driver it uses the new data register of the CAN interface to read message objects whenever new data has been received. The frequency and types of data is hard-coded into the BMS firmware and cannot be configured without intervention from the manufacturer.

3.7. Navigation algorithm

The task of the navigation algorithm is to ensure that the control system can determine the location of the pod and it's velocity.

The primary navigational sensors are optical contrast sensors that recognise optical markings placed in 30m intervals along the track. Although the inverters provide rotational speed measurements from the motors (and therefore the wheels), this on it's own would not be a sufficient. It is not unlikely, that despite the traction controllers influence the wheels could slip during acceleration. In the event of slipping on the wheels the data from the motors becomes useless for navigational purposes. Furthermore, from a safety point of view, it is not necessary to continuously update the pods position or velocity. The control system only needs to know when the pod has exceeded a pre-defined distance in order to engage the brakes at the correct time. Nonetheless, the rotational speed measurements do influence the navigation algorithm to some extent.

Due to the importance of correct navigation and because only one type of sensor is used as a primary input, the system must incorporate fault detection and recovery. The pod has two laser contrast sensors: one at the front and one at the back. This means that at most one laser sensor may miss an optical marking and in almost all situations the system will be able to detect this fault and recover. This is implemented by observing the order in which the sensors trigger. The sensors provide digital output signals that go high when an optical marking is detected, which triggers an interrupt. The following listing shows the interrupt service routine for the front sensor:

```
__interrupt void tape_detection_front_isr(void)
{

    uint64_t now = micros();
    uint64_t min_time_interval = nav_tape_min_time_interval();
```

```
    if (now > tape_detection_time_front + min_time_interval) {

        last_tape_detection_time = tape_detection_time_front;
        tape_detection_time_front = now;
        tape_count_front++;

        // Check if the last tape was missed at the back
        if (tape_count_front > tape_count_back + 1) {
            tape_count_back++;
            tape_miss_count++;
        }

        tape_detected = 1;

    }

    EALLOW;
    PieCtrlRegs.PIEACK.all |= PIEACK_GROUP1;    // ACK the interrupt
    EDIS;

}
```

The interrupt service routine for the back sensor is analogous. By checking if the number of optical markings detected at the front (`tape_count_front`) is at least two greater than at the back (`tape_count_back`), the system can determine if the back sensor missed the previous optical marking.

Should the optical marking be dirty it is possible that the sensor triggers twice or more. To prevent the algorithm from registering this as several markings, it checks if a minimum amount of time has passed since the last trigger. This time period is calculated using a call to `nav_tape_min_time_interval()` where the average of all rotational speed measurements is used to estimate the amount of time the tape sensor is over an optical marking in order to accurately reject false triggers until the optical marking has been passed. The following listing shows this process:

```
uint64_t nav_tape_min_time_interval()
{

    // Get average of wheel speeds
    int32_t speed = -globals.inverter_fl_speed;
    speed += globals.inverter_fr_speed;
    speed += -globals.inverter_bl_speed;
    speed += globals.inverter_br_speed;
```

```
    speed /= 4;

    // Convert speed from rpm to mm/s
    speed *= CONFIG_NAV_WHEEL_CIRCUMFERENCE;
    speed /= 60;

    // Enforce minimum speed
    if (speed < 1000) {
        speed = 1000;
    }

    // Calculate the time interval (us) between the two tape sensors
    uint64_t time = 1000000ull * CONFIG_NAV_TAPE_WIDTH;
    time /= speed;

    // Add a safety margin
    return 8 * time;

}
```

To protect against noise, the input qualification built into the micro-controller is configured to make sure the signal is stable for 15,3 microseconds before it can lead to an interrupt.

Once the time of detection of an optical marking has been recorded, the location and velocity of the pod can easily be calculated and used to make control decisions and modify the global state.

3.8. State Diagram

As a requirement for the competition, the global control algorithm must be implemented as a finite state machine (FSM). The state diagram is checked and tested during the competition. Therefore, the FSM is designed with simplicity in mind and with the least number of automatic transitions in order to reduce the amount of tests necessary to demonstrate correctness.

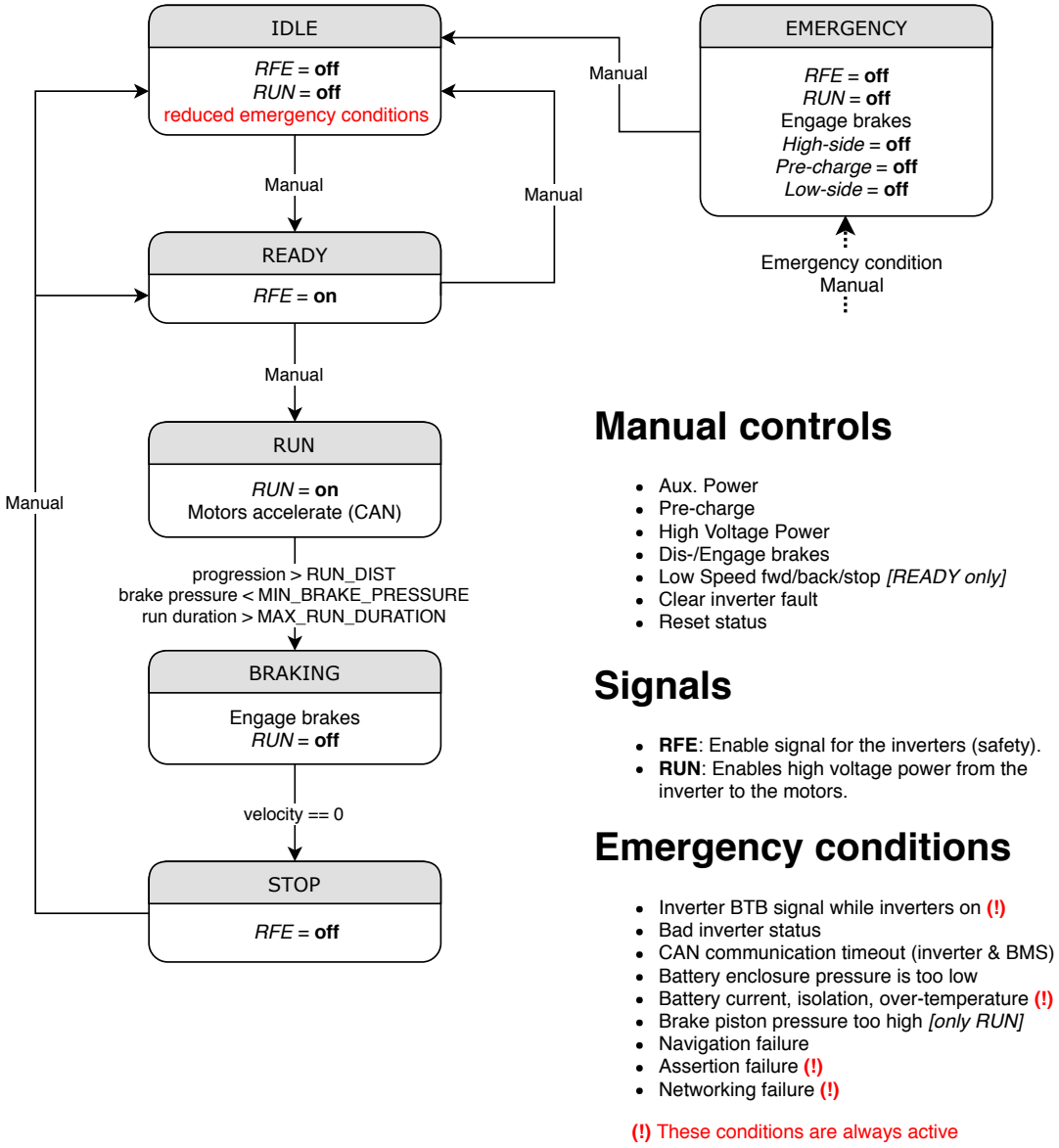


Figure 3.8.: State diagram of the global finite state machine.

Figure 3.8 shows all states and state transitions. Any state transitions that are not listed in Figure 3.8 are illegal and cannot be executed. The `EMERGENCY` state can be entered from any other state automatically if an emergency condition occurs or by a manual command form the control panel. All state transitions labelled "Manual" are also executed using a command from the control panel. Additional safety conditions that may prevent manual state transitions apply in some cases.

The `IDLE` state is intended as the default state when the pod is powered on and stationary. Some emergency conditions are not active in this state as the causes may be cleared by manual intervention with the pod, such as pumping up the brake system to nominal pressure levels.

A nominal run of the pod would consist of the following sequence of states: `READY` → `RUN` → `BRAKING` → `STOP`. In order to begin a run, the system must transition into the `READY` state, insuring that the all emergency conditions are negative and the pod is healthy before proceeding. The `RUN` state represents the acceleration phase of a run and is exited when the pod reaches a pre-defined location, brake failure on at least on of the braking systems appears to be imminent or a pre-defined maximum duration has been exceeded. Once transitioned to the `BRAKING` state, the motors are disabled by turning off the RUN signal and the brakes are engaged. Once the pod has reached a full stop, the pod transitions into the `STOP` state.

Although the FSM covers all safety critical actions, several other controls are left as manual controls to the user in favour of a simpler FSM. However, in several cases automatic safety checks are still in place that prevent manual actions that would lead to unsafe circumstances.

3.9. Control Law Accelerator (CLA)

As mentioned in Section 3.1, two controllers regulating traction and yaw need to be executed at regular intervals. In order to provide an isolated environment and improve performance, these controllers execute on the CLA. The CLA is a co-processor with a separate instruction set that executes in parallel to the CPU at the same frequency. The CLA is configured with up to eight task vectors (function pointers). A task can then be triggered by a peripheral or by software on the CPU. Once triggered, the task runs until it reaches an `MSTOP` instruction.

Blocks of the CPU's dedicated RAM can be configured as program and data space for the CLA. Texas Instruments provides a C compiler for the CLA instruction set. However, the C compiler only supports a subset of C. The main difference being, that the output programs do not use a stack. Instead a scratch-pad concept is used. As a result recursion is not supported. Even so, the compiler is sufficiently capable for control applications.

In order to pass input arguments to the CLA and return outputs to the CPU, the CLA and CPU share a Message RAM consisting of two 128 word segments. One is writeable only by the CPU, the other is only writeable by the CLA. Variables can be allocated in these segments using a `#pragma DATA_SECTION()` directive and then linked in the CPU and CLA code at compile time. Figure 3.9 illustrates this structure.

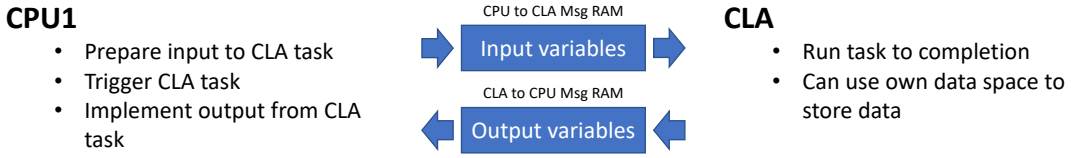


Figure 3.9.: Communication of inputs and outputs for CLA tasks.

3.10. Debugging

During development it can be useful to print messages to a console. For this purpose I configured a third SCI interface on the micro-controller that is connected to a USB-to-Serial converter on the XDS100v2 On-Board Debug Probe of the Launchpad. Using this serial interface the micro-controller can print messages to a host computer. However, in order to make use of support functions such as `printf()` the default Runtime Support Library requires a lot of code and the use of a heap. To avoid these heavy functions I instead used the tinyprintf library[12] which is specifically designed for embedded systems and implements all standard functions such as `printf()`.

Printing messages to the console is synchronous and blocking. However, this is acceptable since printing is not intended for production and is removed in the final build.

While CPU1 can easily output messages over serial, as it is the master of the SCI peripheral, the same is not true for CPU2. Instead of switching the master of the SCI, which would be prone to glitches, CPU2 uses IPC message registers to pass characters to CPU1. An IPC flag is then configured to cause an interrupt on CPU1. The interrupt service routine then prints the character received from CPU2. This mechanism provides both cores with the ability to print debug messages to a console. However, it is important to keep in mind, that this mechanism does not provide any form of locking.

3.11. Unit Testing

In order to ensure correctness, extensive unit tests are employed to test all critical control algorithms and any other modules that can reasonably be tested using software tests. The implemented unit tests specifically also test limits and edge cases that could lead to numerical errors such as integer overflows and underflows. If enabled at compile-time, unit tests execute just before the main loop is entered. However, unit tests are not intended to execute in production, as they may lead to an undesired state and reconfigure some parts of the system for testing.

Chapter 4

Experimental Results

4.1. Performance Evaluation

4.1.1. Logging and SD-card driver

Experimental evaluation of the SD-card driver by visualising write operations using an oscilloscope attached to the SPI bus revealed an important trade-off that greatly impacts the performance of the logging driver. Although many overheads are introduced by the FAT file system, the SD card also introduces an overhead during read and write operations. As write operations are more relevant in this particular application the following will examine only write operations.

Figure 4.1 illustrates the initial steps of writing to an SD-card. First a write command is issued. This is then followed by one or more data packets consisting of a data token, 1-2048 bytes of data and two CRC bytes. Between the cursors in Figure 4.1 one data packet is transmitted. Once all the data packets have been transmitted, a write termination command is transmitted. This last command is omitted if a single block write operation is used.

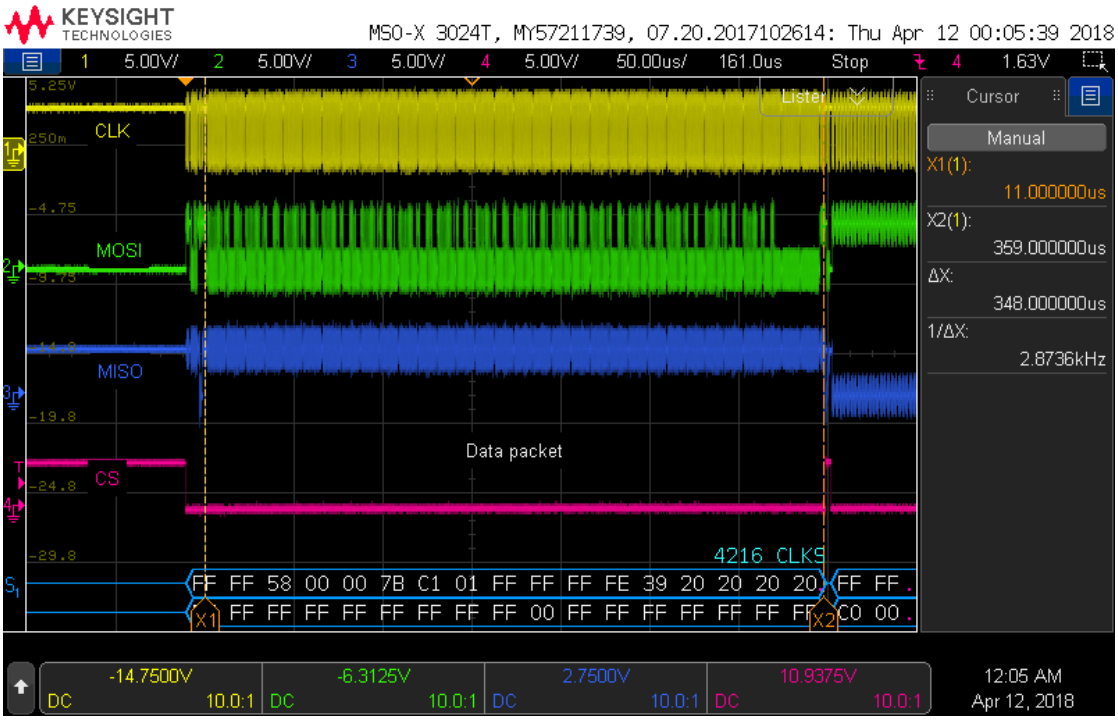


Figure 4.1.: Write command and data packet on SPI bus with SD card.

Between each data packet and after the full sequence of commands and packets, the SD card enters a busy state where it is processing the written data. As Figure 4.2 shows, the busy time after the full sequence (or rather after the write termination command) is particularly long in relation to the transmission time of a single data packet.

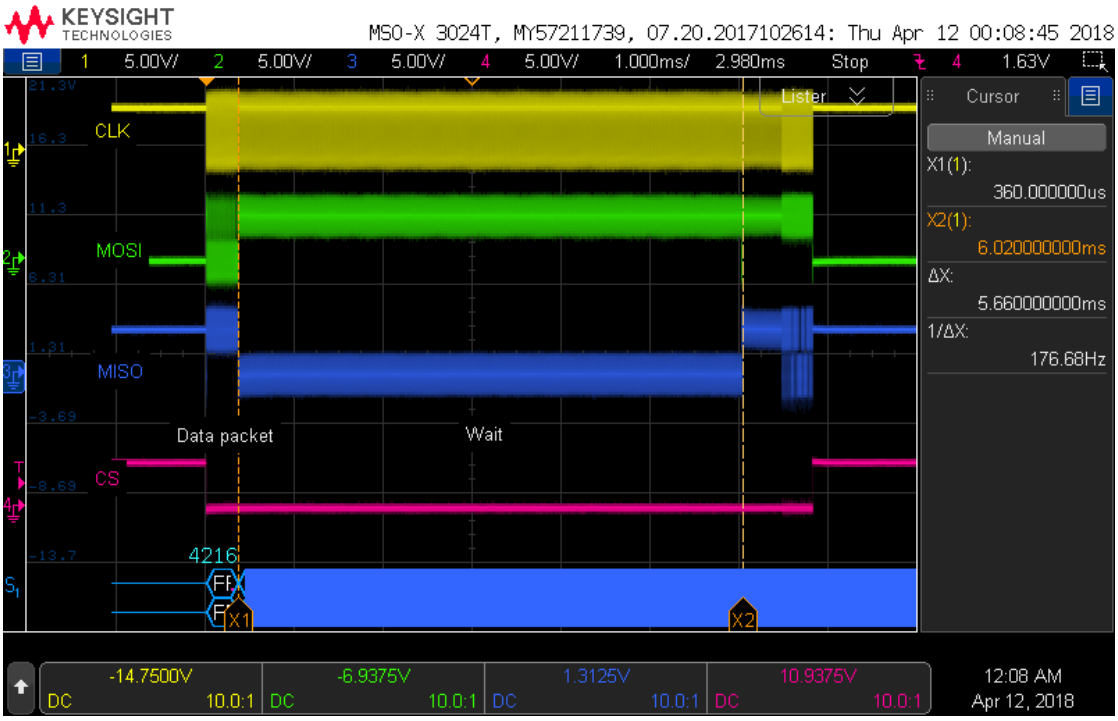


Figure 4.2.: Wait time after writing data packet to SD card.

Further testing revealed that, while the busy time between data packets is very small, the busy time is almost completely dominated by the busy time at the end of a write operation. This is further verified by another test where chunks of different sizes are repeatedly written to a file in the FAT file system and the data rate observed. Table 4.1 shows the results of this test. As can be seen from Figure 4.4 the write speed behaves almost linearly compared to the chunk size.

Table 4.1.: Write speeds for chunks of different sizes.

Chunk size	**Average data-rate**
256 bytes	21 kB/s
512 bytes	41 kB/s
1024 bytes	80 kB/s
2048 bytes	148 kB/s
4096 bytes	254 kB/s

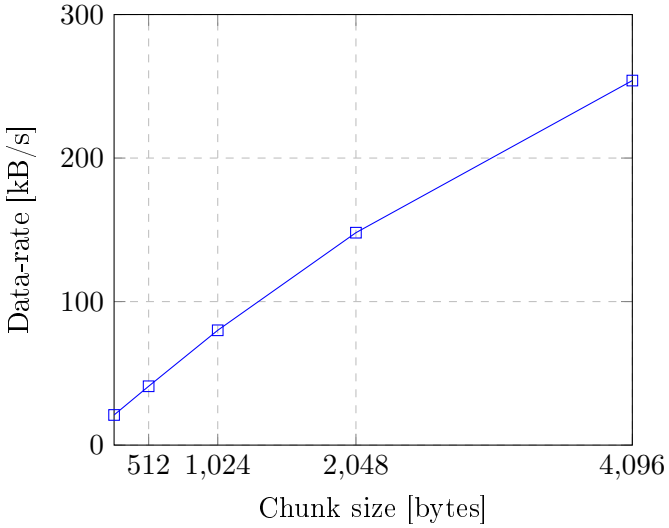


Figure 4.3.: Write speeds for chunks of different sizes.

Given these results we have a memory/data-rate trade-off. Memory is limited on the micro-controller but we also want to maximise the amount of data that can be logged. In-field experimentation showed that under most circumstances write buffers of 2048 bytes are sufficient to achieve the data-rate necessary to log all sensor data. Nonetheless, we ended up using buffers of 4096 bytes to insure that no data would be lost.

Comparison to Escher

The most important objective in comparison to Escher was to improve the logging system. Already by introducing event-based logging more infrequent measurements can be logged less than more frequent measurements. As a result, higher data rates for more important data can be achieved by reducing the bandwidth used by less important measurements. More dramatically, the redesigned system logged on average 1355 data points per second at a data-rate of 29 kB/s, while the system on Escher achieved only a constant rate of 173,5 data points per second.

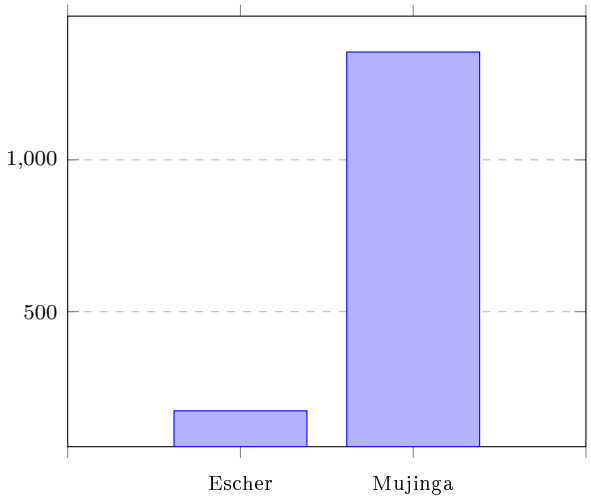


Figure 4.4.: Average number of logged data points per second.

4.1.2. Networking

Using an oscilloscope the SPI bus with the micro-controller and Ethernet controller can be monitored. Figures 4.5 and 4.6 visualise the sequence of commands necessary for sending and receiving short 12 byte network packets. In order to send a packet, the micro-controller must first read the write pointer it should use from the Ethernet controller. Then it can write to the Ethernet controller's packet RAM at the read address. It must then update the write pointer so the Ethernet controller knows the length of the packet and finally confirm the packet using a send command. The sequence for receiving packets is analogous. Notice however, that the process is triggered by the interrupt signal from the Ethernet controller.

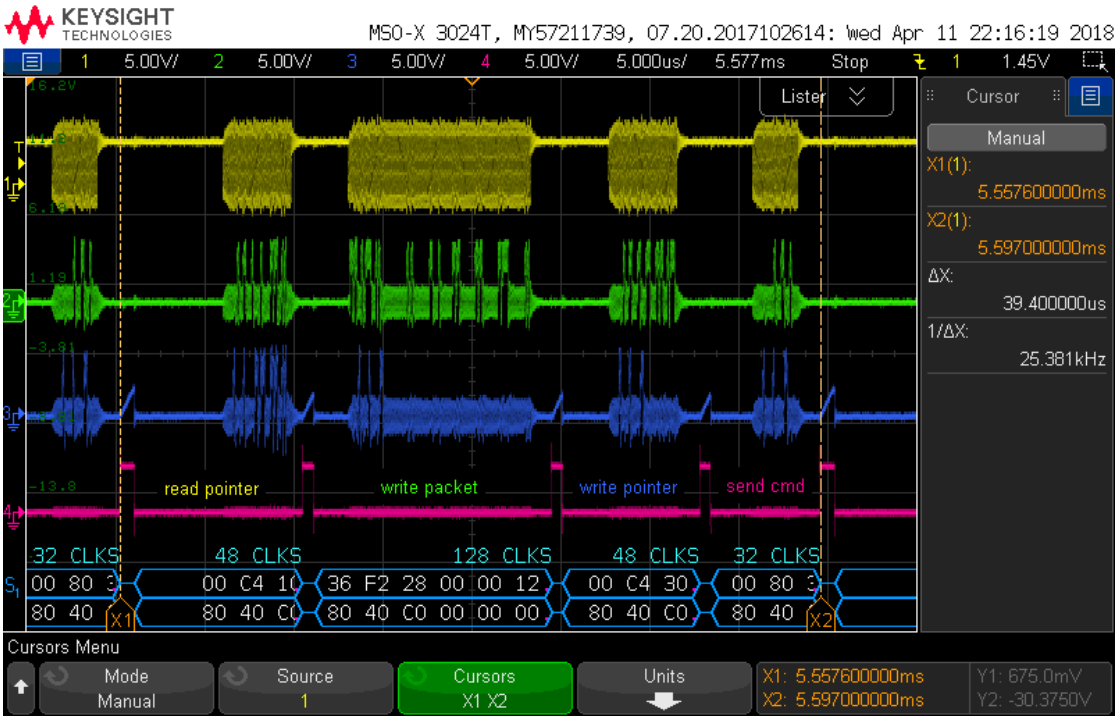


Figure 4.5.: Command sequence for sending a network packet on SPI bus.

From Figure 4.5 we can see that it takes only 39 microseconds to complete all commands necessary to send a network packet of this size at maximum clock rate supported by the SPI interface (12,5 MHz). The use of the DMA controller leads to uninterrupted transmission of each command on the bus. Although the demonstrated performance is more than fast enough for this application, the driver could still be optimised to reduce the time between commands. Some of the computation necessary for the next command is performed after the previous command is transmitted. Although this would increase memory usage, it could be performed while the previous command is still transmitting. On the other hand, for longer packets the time is dominated by the transmission subsequence labelled "write packet" in Figure 4.5.

The same optimisation could be applied to the reception of network packets but again Figure 4.6 shows that the full sequence only takes 47 microseconds, which is sufficient for this application.

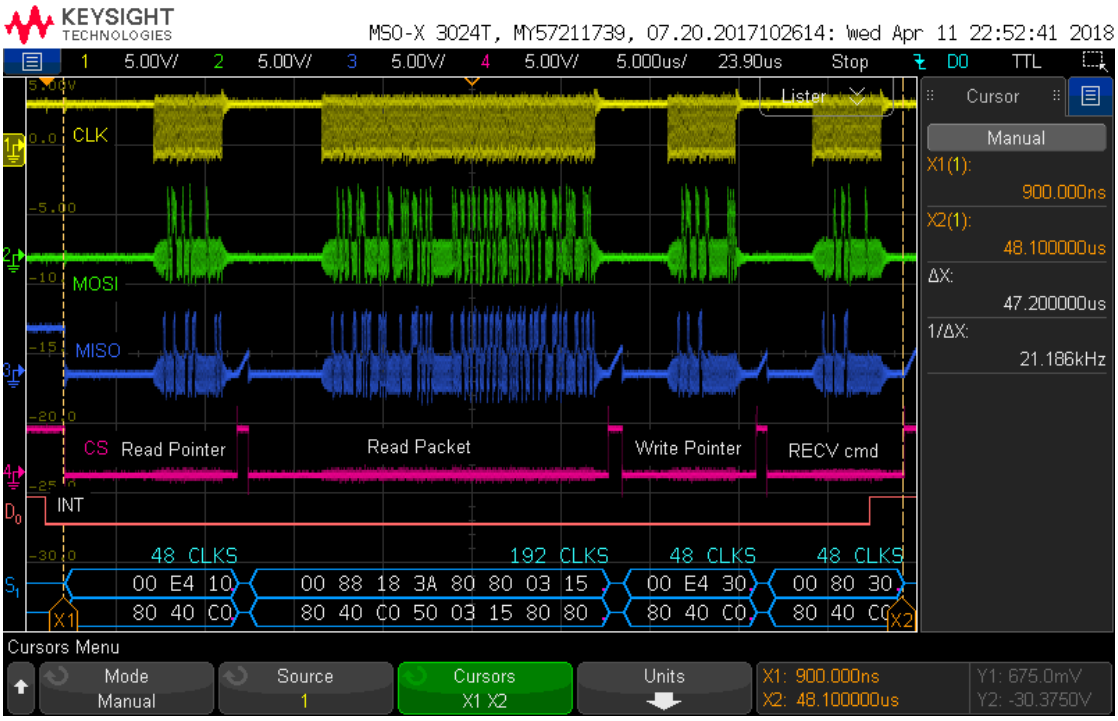


Figure 4.6.: Command sequence for receiving a network packet on SPI bus.

4.2. In-Field Evaluation and Competition

The control system was implemented specifically for the Hyperloop Pod Student Competition. It was extensively tested before the competition during functional tests and on a test 150m test track in Switzerland. During the competition the software was tested in several tests, most notably the functional test, navigation test and state diagram test.

4.2.1. Pre-competition testing

After checking the correct operation of all sensors and actuators in conjunction with the software, we checked that all emergency conditions and safety features worked correctly. Next, full pod operation was tested in a vacuum chamber. We were successfully able to spin up the motors in vacuum and all components operated nominally during the test. The pressurized battery compartments remained pressurized and the cells were therefore not damaged.

In addition to the vacuum chamber test, the team was able to use a 150m test track designed to match the specifications of the test tube at the competition. We completed several runs demonstrating correct operation of the navigational sensors, the navigation

algorithm, the control of the inverters using both speed and torque commands, the control of the brake system and telemetry transmission and logging. Further test runs focused on applying maximal acceleration over short distances in order to fine-tune traction control.

4.2.2. Competition

Throughout the competition week the pod was subjected to a wide range of tests. Three tests specifically target the control software developed in this thesis.

First, the functional test ensures that after power-on the pod is in a safe state and telemetry is received with nominal values. During this test the team also demonstrated correct operation of all safety-critical pod functions such as the brakes from software and using the control panel to command the pod.

Next, the navigation test verifies the navigational mechanisms built into the software work correctly and the level of fault tolerance is assessed. In order to show correct functionality of the navigational algorithm, all possible scenarios were simulated using fake optical markings while the pod remained stationary and the software responses were observed using the telemetry data visualised in the control panel.

Finally, the state diagram test consists of verifying the pod operates according to the state diagram (Figure 3.8). Essentially, this test checks that the global finite state machine is correctly implemented. To demonstrate this, all automatic transitions are tested by simulating all possible causes for the transition. For example, the transition `RUN` $\rightarrow$ `BRAKING` should occur after the pod has travelled a pre-defined distance. This can be simulated by passing fake optical markings in front of the laser contrast sensors.

With all these tests passed, the team had actually gained a decent lead compared to the other teams in the competition. The next test for the pod was the vacuum chamber test, where the full pod is placed in a large vacuum chamber. Although this test had been previously carried out successfully in Switzerland, the pod suffered a sever failure during this test preventing the team from advancing to the finals of the competition.

Shortly after enabling the high-voltage systems in vacuum, a loss of communications occurred. After re-pressurising the vacuum chamber it was immediately possible to see that the low-voltage electronics had been fatally damaged. Further investigation revealed, that the high-voltage batteries had been short circuited. Surprisingly, the low-voltage systems continued to log data for a full second after the failure. After analysing the data, the failure could be narrowed down to one of the inverters. A manufacturing fault caused arcing to occur between the two battery poles. The arcing damaged the inverter, portions of the low-voltage electronics, the low-voltage battery and both high-voltage batteries. Although we were able to repair all other components, the high-voltage battery cells were too severely damaged from the over current to be safely used any further.

4.3. Conclusion

In conclusion, a control system was developed for a prototype Hyperloop pod and deployed at the Hyperloop Pod Competition hosted by SpaceX in Los Angeles, California. All major design goals could be met but some further optimization in some areas is possible. Although the team did not advance to the finals of the competition, the overall system was well designed and will provide a solid foundation for future Hyperloop projects.

Appendix A

Telemetry Frame

```
struct telemetry_frame {

    uint32_t dist_rail_top_front;    // um
    uint32_t dist_rail_top_back;     // um

    uint32_t inverter_fl_status;     // enum
    uint32_t inverter_fr_status;     // enum
    uint32_t inverter_bl_status;     // enum
    uint32_t inverter_br_status;     // enum

    uint16_t state;                  // enum
    uint16_t emergency_reason;       // enum
    uint16_t brakes_engaged;         // boolean
    uint16_t log_file_num;           // number
    uint16_t log_discarded_data;     // boolean
    uint16_t tape_count;             // number
    uint16_t tape_miss_count;        // number

    uint16_t progression;            // m
    uint16_t velocity;               // km/h
    uint16_t max_velocity;           // km/h
    uint16_t run_timer;              // ms

    uint16_t inverter_AUX;           // boolean
    uint16_t inverter_RFE;           // boolean
    uint16_t inverter_RUN;           // boolean
    int16_t inverter_fl_torque;      // %
```

```
int16_t inverter_fr_torque;     // %
int16_t inverter_bl_torque;     // %
int16_t inverter_br_torque;     // %

int16_t inverter_fl_speed;      // rpm
int16_t inverter_fl_motor_temp; // °C
int16_t inverter_fl_igbt_temp;  // °C
int16_t inverter_fl_motor_current;  // A
int16_t inverter_fl_dc_bus;     // V
int16_t inverter_fr_speed;      // rpm
int16_t inverter_fr_motor_temp; // °C
int16_t inverter_fr_igbt_temp;  // °C
int16_t inverter_fr_motor_current;  // A
int16_t inverter_fr_dc_bus;     // V
int16_t inverter_bl_speed;      // rpm
int16_t inverter_bl_motor_temp; // °C
int16_t inverter_bl_igbt_temp;  // °C
int16_t inverter_bl_motor_current;  // A
int16_t inverter_bl_dc_bus;     // V
int16_t inverter_br_speed;      // rpm
int16_t inverter_br_motor_temp; // °C
int16_t inverter_br_igbt_temp;  // °C
int16_t inverter_br_motor_current;  // A
int16_t inverter_br_dc_bus;     // V

uint16_t hv_bat_BMS;            // boolean
uint16_t hv_bat_HS;             // boolean
uint16_t hv_bat_PRE;            // boolean
uint16_t hv_bat_LS;             // boolean

uint16_t hv_bat_l_isolation;    // boolean
int16_t hv_bat_l_voltage;       // V
int16_t hv_bat_l_min_voltage;   // mV
int16_t hv_bat_l_current;       // A
int16_t hv_bat_l_max_cell_temp; // °C
uint16_t hv_bat_r_isolation;    // boolean
int16_t hv_bat_r_voltage;       // V
int16_t hv_bat_r_min_voltage;   // mV
int16_t hv_bat_r_current;       // A
int16_t hv_bat_r_max_cell_temp; // °C

uint16_t lv_bat_current;        // mA
uint16_t lv_bat_voltage;        // mV
```

```
    uint16_t lv_bat_capacity_used;  // mAh

    uint16_t dist_rail_side_front;  // mm
    uint16_t dist_rail_side_back;   // mm

    uint16_t press_brake_block_front;   // bar
    uint16_t press_brake_block_back;    // bar
    uint16_t press_brake_piston_front;  // bar
    uint16_t press_brake_piston_back;   // bar

    uint16_t press_ambient;         // mbar
    uint16_t press_hv_bat_left;     // mbar
    uint16_t press_hv_bat_right;    // mbar

};
```

Appendix B

Logging event ID enumeration

```
enum log_event {

    /*
     * Pod events
     */
    LOG_EVENT_STATE_TRANSITION = 1,      // enum emergency_reason
    LOG_EVENT_EMERGENCY,                 // enum emergency_reason

    /*
     * Inverters
     */
    LOG_EVENT_INVERTER_AUX,              // boolean
    LOG_EVENT_INVERTER_RFE,              // boolean
    LOG_EVENT_INVERTER_RUN,              // boolean
    LOG_EVENT_INVERTER_CLEAR_FAULT,      // -
    LOG_EVENT_INVERTER_FL_SET_TORQUE,    // %
    LOG_EVENT_INVERTER_FR_SET_TORQUE,    // %
    LOG_EVENT_INVERTER_BL_SET_TORQUE,    // %
    LOG_EVENT_INVERTER_BR_SET_TORQUE,    // %
    LOG_EVENT_INVERTER_SET_SPEED,        // %
    LOG_EVENT_INVERTER_FL_STATUS,        // enum
    LOG_EVENT_INVERTER_FL_SPEED,         // rpm
    LOG_EVENT_INVERTER_FL_MOTOR_TEMP,    // °C
    LOG_EVENT_INVERTER_FL_IGBT_TEMP,     // °C
    LOG_EVENT_INVERTER_FL_MOTOR_CURRENT,// A
    LOG_EVENT_INVERTER_FL_DC_BUS,        // V
    LOG_EVENT_INVERTER_FR_STATUS,        // enum
```

```
LOG_EVENT_INVERTER_FR_SPEED,         // rpm
LOG_EVENT_INVERTER_FR_MOTOR_TEMP,    // °C
LOG_EVENT_INVERTER_FR_IGBT_TEMP,     // °C
LOG_EVENT_INVERTER_FR_MOTOR_CURRENT,// A
LOG_EVENT_INVERTER_FR_DC_BUS,        // V
LOG_EVENT_INVERTER_BL_STATUS,        // enum
LOG_EVENT_INVERTER_BL_SPEED,         // rpm
LOG_EVENT_INVERTER_BL_MOTOR_TEMP,    // °C
LOG_EVENT_INVERTER_BL_IGBT_TEMP,     // °C
LOG_EVENT_INVERTER_BL_MOTOR_CURRENT,// A
LOG_EVENT_INVERTER_BL_DC_BUS,        // V
LOG_EVENT_INVERTER_BR_STATUS,        // enum
LOG_EVENT_INVERTER_BR_SPEED,         // rpm
LOG_EVENT_INVERTER_BR_MOTOR_TEMP,    // °C
LOG_EVENT_INVERTER_BR_IGBT_TEMP,     // °C
LOG_EVENT_INVERTER_BR_MOTOR_CURRENT,// A
LOG_EVENT_INVERTER_BR_DC_BUS,        // V

/*
 * Battery Management System
 */
LOG_EVENT_BMS_LS,                // boolean
LOG_EVENT_BMS_PRE,               // boolean
LOG_EVENT_BMS_HS,                // boolean

/*
 * High voltage batteries
 */
LOG_EVENT_HV_BAT_L_VOLTAGE,          // V
LOG_EVENT_HV_BAT_L_MIN_VOLTAGE,      // mV
LOG_EVENT_HV_BAT_L_CURRENT,          // A
LOG_EVENT_HV_BAT_L_MAX_CELL_TEMP,    // °C
LOG_EVENT_HV_BAT_R_VOLTAGE,          // V
LOG_EVENT_HV_BAT_R_MIN_VOLTAGE,      // mV
LOG_EVENT_HV_BAT_R_CURRENT,          // A
LOG_EVENT_HV_BAT_R_MAX_CELL_TEMP,    // °C

/*
 * Brake event
 */
LOG_EVENT_BRAKES_ENGAGED,        // boolean

/*
```

```
 * Navigation
 */
LOG_EVENT_TAPES_DETECTED,        // number
LOG_EVENT_TAPES_MISSED,          // number
LOG_EVENT_PROGRESSION,           // mm
LOG_EVENT_VELOCITY,              // mm/s

/*
 * Low voltage battery
 */
LOG_EVENT_LV_BAT_CURRENT,        // mA
LOG_EVENT_LV_BAT_VOLTAGE,        // mV
LOG_EVENT_LV_BAT_CAPACITY_USED, // mAh

/*
 * OM70 laser sensors
 */
LOG_EVENT_DIST_RAIL_SIDE_FRONT, // um
LOG_EVENT_DIST_RAIL_SIDE_BACK,  // um

/*
 * OADM laser sensors
 */
LOG_EVENT_DIST_RAIL_TOP_FRONT,  // um
LOG_EVENT_DIST_RAIL_TOP_BACK,   // um

/*
 * PBM4 pressure sensors
 */
LOG_EVENT_PRESS_BRAKE_BLOCK_FRONT,  // bar
LOG_EVENT_PRESS_BRAKE_BLOCK_BACK,   // bar
LOG_EVENT_PRESS_BRAKE_PISTON_FRONT, // bar
LOG_EVENT_PRESS_BRAKE_PISTON_BACK,  // bar

/*
 * PBMN pressure sensors
 */
LOG_EVENT_PRESS_AMBIENT,         // mbar
LOG_EVENT_PRESS_HV_BAT_LEFT,     // mbar
LOG_EVENT_PRESS_HV_BAT_RIGHT,    // mbar

/*
 * Controller outputs
```

```
     */
    LOG_EVENT_YAW_MOMENTUM = 400,    // mNm
    LOG_EVENT_TRACTION_TORQUE_FL,    // number
    LOG_EVENT_TRACTION_TORQUE_FR,    // number
    LOG_EVENT_TRACTION_TORQUE_BL,    // number
    LOG_EVENT_TRACTION_TORQUE_BR,    // number

};
```

Appendix C

Declaration of Originality

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Multi-Sensors Control System for a Transportation Vehicle in a Low-Pressure Environment

Authored by (in block letters):

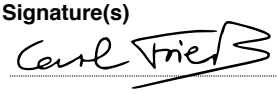
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Friess	Carl

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 26 August 2018	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Bibliography

[1] "Integrated Systems Laboratory, Department of Information Technology and Electrical Engi- neering, ETH Zurich," [accessed 26-August-2018]. [Online]. Available: http://www.iis.ee.ethz.ch

[2] "SN65HVD7x 3.3-V Supply RS-485 With IEC ESD Protection," [accessed 26-August-2018]. [Online]. Available: http://www.ti.com/lit/ds/symlink/sn65hvd72.pdf

[3] Elon Musk (SpaceX), "Hyperloop alpha," 2013, [accessed 22-August-2018]. [Online]. Available: https://www.spacex.com/sites/spacex/files/hyperloop_alpha.pdf

[4] "Spacex hyperloop pod competition," [accessed 22-August-2018]. [Online]. Available: https://www.spacex.com/hyperloop

[5] "Swissloop," [accessed 26-August-2018]. [Online]. Available: https://swissloop.ch

[6] "C2000 Delfino MCUs F28379D LaunchPad Development Kit," [accessed 26-August-2018]. [Online]. Available: http://www.ti.com/tool/LAUNCHXL-F28379D

[7] "TMS320F28379D Dual-Core Delfino Microcontroller," [accessed 26-August-2018]. [Online]. Available: http://www.ti.com/product/TMS320F28379D

[8] "controlSUITE™ Software Suite: Software and Development Tools for C2000™ Microcontrollers," [accessed 26-August-2018]. [Online]. Available: http://www.ti.com/tool/CONTROLSUITE

[9] "log_fs.h : Simple record-based file system for Serial Flash," [accessed 26-August-2018]. [Online]. Available: http://piconomix.com/fwlib/group___l_o_g___f_s.html

[10] "Yaffs (Yet Another Flash File System)," [accessed 26-August-2018]. [Online]. Available: https://yaffs.net

[11] "FatFs - Generic FAT Filesystem Module," [accessed 26-August-2018]. [Online]. Available: http://elm-chan.org/fsw/ff/00index_e.html

[12] "tinyprintf - a tiny printf and sprintf library for small embedded systems," [accessed 26-August-2018]. [Online]. Available: https://github.com/cjlano/tinyprintf