TU Delft

Design of an Electronic Speed Controller

Sub-group: Communication and Sensing

B.Sc. Thesis
by Ruben Vos and Quinten Luyten

15 June 2023

DELFT UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE

ELECTRICAL ENGINEERING PROGRAMME

This thesis has been prepared with contributions from:

	Student numbers	5244803 (Quinten Luyten)
		5268966 (Ruben Vos)
	Fusion Engineering	Project proposer
Thesis comittee	Dr. Jianning Dong	TU Delft Supervisor
	Prof. Dr. Andrea Neto	
	Prof. Mansureh Shahraki Moghaddam	

Abstract

This thesis is part of a project where a 120A, 50V (12S Lithium-Polymer battery) electronic speed controller (ESC) with Bidirectional DShot600 support is prototyped. This thesis focuses on the communication and sensing aspects of the ESC. It discusses the implementation of the DShot communication protocol on an STM32G431KBT6 microcontroller. Also, the microcontroller senses phase currents and phase voltages from a BLDC motor and has to provide all the necessary information and processing resources for a control algorithm to be implemented. A PCB is designed to break out all the required interfaces of the microcontroller. Lastly, temperature management is implemented to prevent overheating of the inverter hardware by means of throttling. The DShot implementation has full bidirectional functionality with 3.4% package corruption. All sensor values are accurately provided to the control algorithm, and temperature and speed information are provided as DShot telemetry. Due to time constraints and lack of integration with the control algorithm, the behavior during anomalies such as power interrupts is not yet thoroughly tested.

Preface

This thesis was drawn in the context of the Bachelor Electrical Engineering Graduation project (BAP) for the graduation year 2023 at the TU Delft. The thesis was made in close collaboration with the other subgroups working on the Electronic Speed Controller.

We would like to thank Dr. Jianning Dong for his advice and supervision of the project. We are very thankful to Fusion Engineering for providing the project proposal, funding, and advice on the implementation of the DShot interface. We would also like to thank Ron van Puffelen for his advice on PCB design, and the DCE&S Lab staff for providing test fixtures.

Contents

Glossary of frequently used terms

ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuit
BEMF	Back Electromotive Force - A voltage induced over the motor windings due to the rotation of the rotor magnets
BLDC	Brushless DC motor - a synchronous three-phase motor with permanent magnets
Control	The algorithm that controls the inverter bridges based on sensor data
Controller	The integrated circuit(s) responsible for logic, control, and communication
CORDIC	Coordinate rotation digital computer - a trigonometric algorithm
CRC	Cyclic Redundancy Check
DAC	Digital to analog converter
DShot	Digital ESC communication protocol used on drones
DSP	Digital Signal Processor, a kind of microcontroller aimed at signal processing and control
eRPM	Electrical rotations per minute - $60 \cdot f_{electrical}$
EMI	Electromagnetic Interference
ESC	Electronic Speed Controller - A motor controller meant for small, high speed 3-phase motors
flight controller	The main processor on an aircraft that processes sensor data and user commands [1]
FOC	Field-Oriented Control - motor control algorithm
FPGA	Field Programmable Gate Array
GCR	Group Coded Recording - A method for transmitting serial data through a bandwidth limited channel
GPIO	General Purpose Input/Output pin on an MCU or FPGA
JTAG	Joint Test Action Group - A programming and debugging interface for MCUs and FPGAs
IDE	Integrated Development Environment
LiPo	Lithium Polymer battery
MCU	Microcontroller Unit - a processor (integrated circuit) for embedded applications
MSB first	The Most Significant Bit is transmitted first in time
PCB	Printed circuit board - fibreglass plate with circuit traces
PMSM	Permenant Magnet Synchronous Motor
SPI	Serial Peripheral Interface - a high speed communication interface
SMD	Surface mount device - electronic components mounted on a PCB
SWD	Serial Wire Debug - A programming and debugging interface for ARM Cortex MCUs
USART	Universal Synchronous or Asynchronous Receiver Transmitter

Chapter 1

Introduction

1.1 Problem definition

The bachelor graduation project topic was proposed by Fusion Engineering [2]. The company is in need of a high-power ESC (Electronic Speed Controller), since the current high-power ESCs on the market are unreliable or lack specific features. The following points are the main objective of the project, with only the first two being applicable to this thesis.

- The ESC should support DShot600, which is a digital protocol designed for communication between flight controllers and ESCs on drones.
- Since reliability is key, the ESC should transmit emerging problems such as overheating back to the flight controller, such that appropriate action can be taken.
- The ESC must operate at a supply voltage from 36 up to 50.4 volts and draw 120A continuously.

The main constraints limiting the project are time and budget constraints. The ESC design and its respective thesis should be completed in two months, with a limit to the cost of components for prototyping.

1.2 System Overview

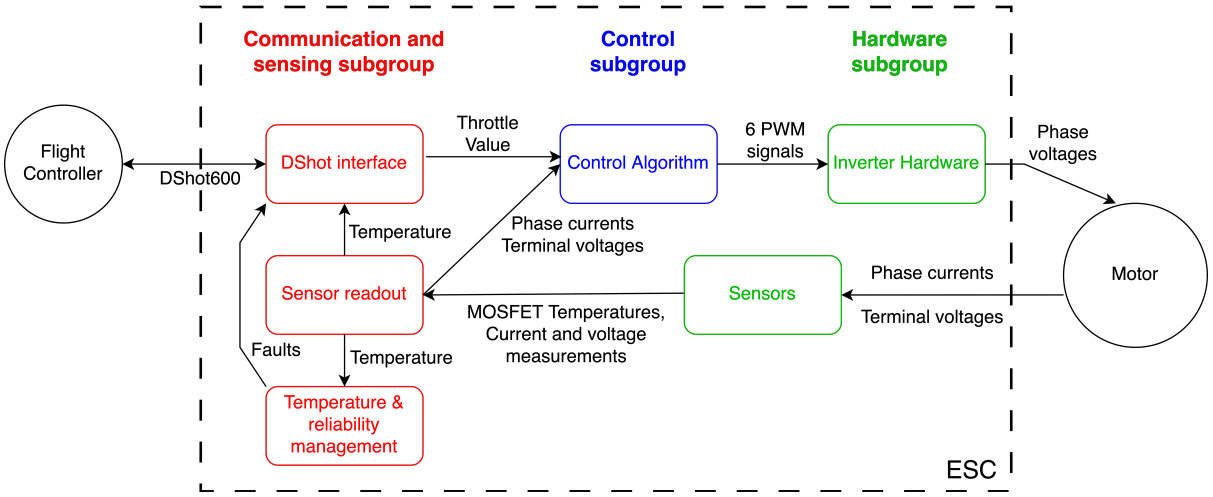


Figure 1.1: Overview and division of the ESC project

The ESC consists of a communication interface, a control algorithm, and a hardware design. Figure 1.1 shows how the project is divided into modules. The first modules, shown in red, fall under the responsibility of the communication and sensing subgroup. This module provides the interface between the flight controller and the ESC. It also facilitates communication between the different modules within the ESC. These are the control module, shown in blue, and the inverter module, shown in green. The control module is responsible for driving

the inverter in such a way that the motor can reach the speed that is commanded by the flight controller. This is done by sending the duty cycles to the inverter. The design and assembly of the inverter is the task of the hardware subgroup. This subgroup needs to make sure that the inverter can operate at the required currents and voltages. They also do current and voltage measurements that are necessary for the control algorithm. Temperature measurement is also to be implemented by this subgroup, which is referred back to the flight controller through the communication interface.

1.3 Synopsis

The thesis can be divided into two main parts. Firstly, it describes the design process of a high-power ESC controller. Secondly, it discusses the testing of the communication algorithm and the accuracy of digitization of sensor readings.

Design requirements are set in Chapter 2. From those, a controller is chosen that fits the requirements (Chapter 4). For that controller, a PCB (Printed Circuit Board) is designed to make all the necessary features available (Chapter 5). Next, Chapter 6 discusses the software implementation of these features. Also, Chapter 3 gives a precise definition of DShot.

Figure 1.2 shows the breakdown of the communication and sensing submodules and how they are integrated. The DShot line is the physical wired connection between the flight controller and the ESC. It interfaces with the DShot transmitter and receiver on the ESC (Section 6.1). These modules provide the interface between the physical signal and the software that runs on the controller. Also, the necessary sensor values are read through ADCs (Section 6.3) and an SPI interface (Section 6.4).

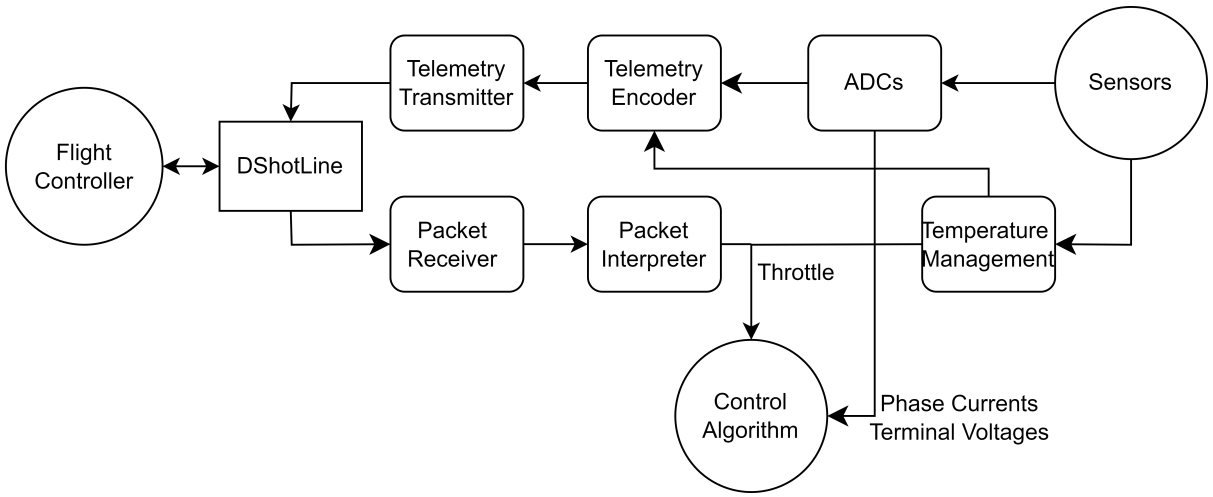


Figure 1.2: Overview of how the communication and sensing modules interface with each other

An important part of the discussion is the effectiveness of the DShot algorithm compared to more common protocols. Therefore, the error rate and CPU utilization of the DShot implementation are tested. Also, the ADCs are tested to estimate their accuracy, since the control algorithm requires accurate data to be effective. The ADCs have an LSE of 0.25% and a maximal integral non-linearity (INL) of 2.1% (Section 7.3).

1.4 State-of-the-art Analysis

1.4.1 Analog communication protocols

Different communication protocols are used on drones to transmit data between the flight controller of the aircraft and the motor controllers. Of these, DShot has quickly become the most popular [3] because of reasons explained in Section 1.4.2.
PWM, Oneshot, and Multishot are analog protocols that all use the duty cycle of a data packet as a direct measure of the requested throttle value. These protocols are not thoroughly discussed, but can be a useful alternative to DShot if problems occur during implementation.

PWM has a very low update rate. Oneshot and Multishot are defined in the same way as PWM, but with shorter pulse lengths, which means that the maximal update rate can be higher.

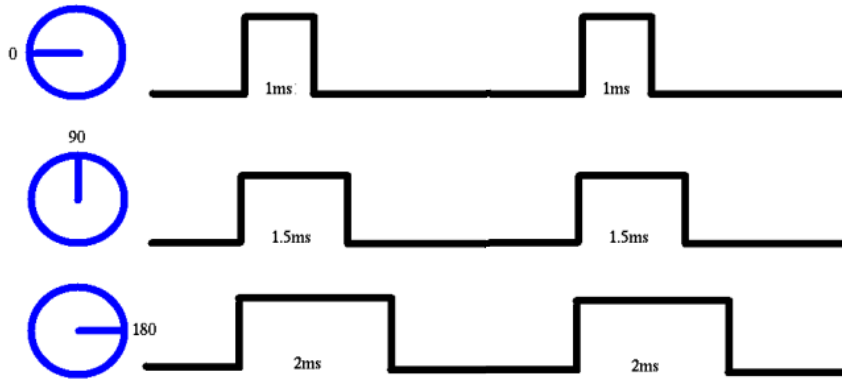


Figure 1.3: PWM protocol. The protocol is often used to indicate angles for servo motors, but can also be used to transmit throttle values.

1.4.2 DShot

DShot, short for Digital Shot, is a standard communication protocol for ESCs first introduced in the Betaflight flight control firmware [4]. The protocol sends the throttle value calculated in the flight controller to the ESC. The biggest difference between DShot and other protocols is that DShot is digital, which has significant benefits in comparison to the analog counterparts. These benefits include:

- Error checking, DShot has bits reserved to guarantee to a reasonable degree that the data is or is not corrupted in transmission.
- More accurate throttle values
- Possible two-way communication on one wire
- Precise communication requests
- Throttle values received are not noisy which ensures more stable motor control

There are also downsides to using DShot:

- When a bit is corrupted, the entire package is thrown out which can cause the ESC to be unable to be communicated to, while in the analog version, the data just gets noisy. This means that in highly noisy areas, communication could get difficult.
- Although DShot is fast, it does not use its available bandwidth optimally, since time is used for error checking and radio-silence between packages.
- If the connection between the flight controller and the ESC is long and/or of low quality, there can be too much capacitance on the line, which causes an increase in the slew rate. Because of the high bit rate of DShot, bits can get corrupted if the slew rate is too large. This problem affects all digital communication protocols.

The DShot algorithm is already implemented in many ESCs. For example, the open source BLHeli [5] and Bluejay [6] firmware have full support for DShot. The precise definition of DShot is given in Chapter 3

1.4.3 Available Controllers

FPGA

An FPGA (Field Programmable Gate Array) consists of arrays of programmable logic gates. Their main advantage over MCUs or DSPs is the amount of parallelism. In an FPGA, different computations can be done in parallel, whereas single-core MCUs can only execute one function at a time [7].

MCU

MCUs or Microcontroller Unit is a CPU with volatile memory, storage, and peripheral interfaces integrated into one chip. They are usually programmed using C(++) or assembly and come with many features found in embedded devices such as ADCs (Analog to Digital Converters), timers, hardware modules for I2C, SPI, and UART. Some MCUs even have multiple cores.

Hybrid approach

Both MCUs and FPGAs have their benefits. FPGAs are better at real-time calculations and performing highly repetitive tasks [8]. MCUs are better at performing changing calculations, such as determining parameters through complex arithmetic [8]. It is possible to split the tasks accordingly, with an FPGA functioning as the DShot decoder and the pulse generator, and a DSP or MCU used for the control algorithm. However, the drawbacks of this approach are a larger product size, since it needs to house two controllers, and possibly a higher cost. This higher cost can be offset by lower individual needs of the components: a very small FPGA and MCU are sufficient in the hybrid approach.

Trigonometric Calculations

For some control algorithms, a lot of trigonometric calculations have to be performed. To meet this requirement, the controller must be able to perform these calculations. These trigonometric calculations require a lot of hardware in both an FPGA and an MCU [9]. In an MCU, it takes many clock cycles to calculate a simple sine if there is no hardware module for it. In an FPGA, a lot of area needs to be reserved for the operation since it requires a large algorithm. When no precise values are needed, estimations can reduce clock cycles or area for the trigonometric calculations. A simple estimate is a lookup table, in which precise results for inputs of a trigonometric operation are stored. If the controller wants to perform a trigonometric operation, it can look at the table and linearly interpolate between the two nearest values.

1.4.4 Sensor Readout

For stable motor control, sensor readouts are needed. These include but are not limited to phase voltages and phase currents. Besides this, it is also useful to have temperature readouts to prevent component breakdowns.

Current measurement

From the hardware subgroup, it's determined that analog sensors are cheaper and smaller. To let the controller interpret this, the analog signal has to be converted to DC. Various ADCs are available on the market. Certain inverter algorithms introduce harmonic distortion in the phase currents, which introduces the need for a low-pass filter before the ADC [10]. Errors in the current measurements, such as an offset or a scaling error, can propagate through the control loop and cause significant torque ripple [11]. The time delay caused by the processing of the current measurement can also impact the control loop [12].

Temperature measurement

For temperature measurement, there is no risk of errors propagating through the control loop, but the accuracy of the measurements is still important to prevent overheating.

Chapter 2

Design Requirements

The company Fusion Engineering tasked us with designing an Electronic Speed Controller (ESC) that can supply 120 amps from a 12s Li-Po battery. It should communicate with the flight controller using the Bi-directional DSHOT600 protocol, and work for the motors they use in their projects. Fusion Engineering prioritizes reliability over everything else, since ESC failures can cause an aircraft to crash. Efficiency is nice to have, but not a priority. The ESC will operate in a low airflow environment in the body of the aircraft. The ESC must function with the "T motor mn501-s 240kv" PMSM motor.

2.1 Electrical requirements

- The ESC must operate at a supply voltage from 36V up to 50.4V.
- The ESC must be able to draw 120 A continuously from the DC bus.

2.2 DShot and telemetry requirements

- The controller must be fully compatible with the extended bidirectional DShot 600 protocol.
- The current consumption and temperature of the ESC must be successfully received by the flight controller at least every 100 ms as DShot telemetry data.
- The telemetry data (electrical period in µs) must be successfully received by the flight controller at least every 1 ms.
- The ESC must be able to operate at the required bandwidth when 90% of DShot packages are corrupted in transmission.

2.3 Speed requirements

- The ESC must at least have an operating electrical frequency range from 0 up to 2333 Hz.

2.4 Fault protection requirements

- The controller must be able to detect when the temperature of the MOSFETs approaches their rated operating temperature and indicate the problem to the flight controller using DShot telemetry.
- The controller must actively prevent the temperature of the MOSFETs from exceeding their operating temperature.
- The ESC must be able to restart without intervention after power interrupts of any duration.
- The ESC must be able to run continuously at its rated supply current and voltage for at least 30 minutes.

- The communication interface should forward any of the following anomalies detected by the control algorithm to the flight controller using DShot telemetry:
  - Sequences of faulty measurements
  - Unexpected changes in speed

2.5 Financial and organizational requirements

- The bill of materials of the final prototype must not exceed €100.
- The total cost of prototype orders, such as components and PCBs, must not exceed €500.
- The bill of materials of the final prototype hardware used by the communication subgroup should be below €30.

2.6 Planning requirements

- The thesis must be completed by the 15th of June.
- Each subsystem must be fully working by the 30th of June

2.7 Interoperability requirements

- The controller must supply a minimum switching speed of 100 kHz.
- The controller must run on a supply of 3.3 V.
- The controller must read one temperature measurement, transmitted over SPI to the controller.
- The controller must read three-phase current measurements, transmitted as three analog signals with a range from 0 to 3.3 V.
- The controller must read three-phase voltage measurements, transmitted as three analog signals with a range from 0 to 3.3 V.
- The controller must supply the phase current and phase voltage measurements available to the control algorithm as digital values.
- The digital values must not have more than 0.25 % full range output linearity error with respect to the best fit straight line over the entire signal range at room temperature.
- The controller must supply the throttle requested by the flight controller to the control algorithm as a digital value.
- The controller must supply the six switching signals from the control algorithm to the gate drivers.

Chapter 3

Definition of the DShot protocol

DShot has a 16 bits frame in the following order:

- 11 bits, for throttle values
- 1 bit, for requesting telemetry
- 4 bits, for error checking using CRC (Cyclic Redundancy Check)

With 11 bits for throttle, a total of 2048 possible values can be used. The value ’0’ is reserved for NOP and the values ’1’ until ’47’ are used for special commands [13].

To send the bits over the line, a ’0’ and a ’1’ need to be distinguished from each other and from silence. This is done by having different duty cycles. The duty cycle of a ’1’ is always double that of a ’0’. The numbers that are declared after DShot equal the bit rate in kbits, so DShot300 has a bit rate of 300kbit/s. The precise timings can be found in Table 3.1 [13].

Table 3.1: Data Timings of different DShot versions [13]

DSHOT	**Bit rate**	**T1H (µs)**	**T0H (µs)**	**Bit (µs)**	**Frame (µs)**
150	150kbit/s	5.00	2.50	6.67	106.72
300	300kbit/s	2.50	1.25	3.33	53.28
600	600kbit/s	1.25	0.625	1.67	26.72
1200	1200kbit/s	0.625	0.313	0.83	13.28

This would mean that the maximal update rate of the ESC is capped by the frame length. However, the control algorithm on the flight controller of a quadcopter usually has a slower update rate than the maximal speed of DShot.

3.1 Bidirectional DShot

Bidirectional DShot lets the ESC send back an “eRPM” telemetry package between the messages sent by the flight controller. This telemetry package contains the time of one electrical period in microseconds. It lets the flight controller keep accurate control of the rotational speeds of the different motors. In Bidirectional mode, the DShot signal is completely inverted, meaning that the duty cycle is considered when the signal is low. Also, the CRC is the inverse of the CRC used for standard DShot.

Since telemetry is sent back between each message, the rate of frames that can be sent is halved. The telemetry bit is still used to request extra telemetry through another line. The eRPM frame that is sent back to the flight controller has the following bit structure:

- 3 bits, Exponent, amount that the mantissa value needs to be shifted to the left.
- 9 bits, Mantissa, value for the electric period in µs.
- 4 bits, Error detection code. The CRC is the inverse of the CRC used for standard DShot.

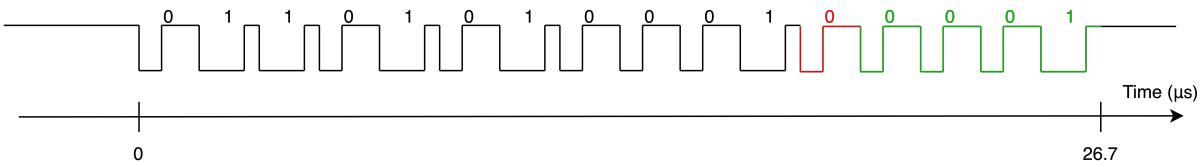


Figure 3.1: Bidirectional DShot600 throttle frame. The throttle requested by the flight controller is “01101010001” (decimal 849). The data is shown in black, the telemetry bit in red and the CRC bits in green.

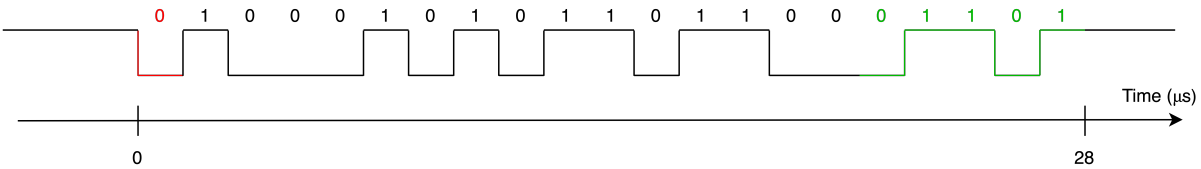


Figure 3.2: DShot telemetry frame using the differential GCR (Group Coded recorded) encoding [13]. The transmission scheme is also different from the throttle packages. The red bit is the start bit, the black bits contain the telemetry data, and the green bits encode the CRC. The decoded package is “0x0c8b”: Exponent = 0, Mantissa = 200, CRC = “1011”.

Telemetry type	Header
Temperature	0010
Voltage	0100
Current	0110
Debug 1	1000
Debug 2	1010
Stress	1100
Status	1110

Table 3.2: The different extended telemetry types and their prefix [14]

It must be noted that this 16-bit telemetry frame is first encoded with differential GCR encoding before being transmitted [13], making it a 21-bit long package.

3.2 Extended Telemetry

In the Extended Telemetry version of DShot, the Bidirectional DShot protocol is further expanded by replacing some eRPM frames with other telemetry data, such as temperature or current measurements done by the ESC [14]. Every telemetry type receives its own 4-bit prefix (Table 3.2, after which a value can be appended. The packages are then processed like normal eRPM frames as explained in Section 6.2.

Chapter 4

Design Considerations

4.1 Choice of controller

A motor controller contains an inverter that needs to be driven with precisely timed digital signals. Also, phase currents and voltages need to be measured for the control algorithm to determine the state of the motor (Section 2.7). Chapter 1.4.3 lists the currently available devices that satisfy these requirements. The following table summarizes the features of the different options:

	FPGA	Microcontroller	ASIC
Main features	Programmable parallel logic	Embedded CPU	High-volume application specific hardware
Parallelism [15]	Easy	Dependent on hardware modules	Easy
Timing precision	Single clock cycle	Dependent on instruction execution	Single clock cycle
Analog interfaces	External	Built-in	Flexible
Hardware Implementation	Multi-Chip [16]	Single-Chip	Time-consuming
Hardware Development time [15]	Weeks	Days	Months
Trigonometric calculations	Consumes lots of available space [9]	Lots of Clock Cycles or dedicated built-in hardware[17]	Consumes lots of available space [9]

Table 4.1: Comparison of FPGAs, Microcontrollers, and ASICs

Due to the prohibitively large development cost, both in time and money, an ASIC is not a viable option for the development of the motor controller (Section 2.5 and 2.6). FPGAs offer many benefits over microcontrollers but suffer from a more expensive and difficult hardware implementation [18]. FPGAs need an external flash chip, since they usually do not have built-in configuration memory [15]. Also, they are more expensive than similarly capable microcontrollers and rarely have integrated analog components such as ADCs, DACs and comparators. Due to the severe time limitations on the project, and the previous experience of the authors with microcontrollers, it was decided to implement a microcontroller on the ESC.

4.2 Choice of microcontroller series

The following requirements have an impact on the choice of the microcontroller:

- The ESC must be able to communicate using the extended Bidirectional DShot 600 protocol (Section 2.2).
- The controller must supply a minimum switching speed of 100 kHz (Section 2.7).
- The controller must run on a supply of 3.3 V (Section 2.7).
- The controller must read a temperature measurement, transmitted over SPI to the controller (Section 2.7).

- The controller must read three phase current measurements, transmitted as three analog signals with a range from 0 to 3.3 V (Section 2.7).
- The controller must read three phase voltage measurements, transmitted as three analog signals with a range from 0 to 3.3 V (Section 2.7).
- The digital values must not have more than 0.25 % full range output linearity error with respect to the best fit straight line over the entire signal range at room temperature (Section 2.7).
- The controller must supply the requested eRPM available to the control algorithm as a digital value (Section 2.7).
- The controller must supply the six switching signals from the control algorithm to the gate drivers (Section 2.7).
- The bill of materials of the prototype hardware used by the communication subgroup should be below €30 (Section 2.5).

In contrast with FPGAs, which are only manufactured by a few companies, there is a very wide selection of microcontroller manufacturers. It is impossible to compare all of them in this section, which is why we focus on only one series. The STM32 series of microcontrollers are commonly used in the hobby quadcopter space [1], which makes it an obvious consideration for the controller, with lots of tutorials and community support. ST Microelectronics also provides clear implementation manuals [17] and datasheets. Other microcontrollers, such as the SiLabs EFM8 [19] or Atmel Atmega series, have relatively low performance, since they are 8-bit MCUs with relatively simple timers and other peripherals. This means that prototype code has to be more efficiently written in order to function properly, since timing is an important requirement of the microcontroller. Digital Signal Processors such as those supplied by Texas Instruments are very well suited for the application at hand, but are relatively expensive [20].

Plenty of high-resolution analog pins are needed to satisfy the requirements for the microcontroller. At least 9 bits of resolution are needed to reach the linearity requirement: $\log_2 0.25\% = -8.644$. The STM32 offers 2 ADCs of 12-bit resolution, with up to 12 channels for each ADC [21]. All STM32 microcontrollers operate on a single supply voltage of 3.3 Volts and have at least 26 GPIO pins, which makes them easy to implement with enough connectivity for all the required interfaces of the motor controller.

Two features which set the STM32G431 apart are the inclusion of a trigonometric processing unit and its low price. The trigonometric unit uses the CORDIC algorithm [21] and operates independently of the main processing core. The manufacturer's recommended retail price ranges from $2 to $5, placing it well within the budget constraints (Section 2.5, even when accounting for the cost of the PCB and all other required passive components. The 32-pin variant of this chip is easier to route on a PCB and has a larger pin pitch, allowing for easier soldering and testing. Finally, since the 32-pin package (type K) offers enough pins for the application at hand, this package was chosen.

For the DShot protocol, the time between two edges needs to be measured. For this, the MCU must have enough clock cycles to process every edge. Next to executing the code for the edge detection, the MCU must also run the control algorithm. The STM32G431 has a clock frequency of up to 170MHz. DShot600 is being used, which means that the bit rate equals 600 kHz (Chapter 3). Each bit has two edges, which means that the edge frequency equals 1.2 MHz. This MCU has therefore 140 clock cycles available to process every signal edge. If it is assumed that the control algorithm will take 50% of those cycles then the interrupt has 70 clock cycles to execute, which should be sufficient.

Other 32-bit microcontrollers with sufficiently advanced peripherals which could fit the project requirements are the NXP LPC series, which are also based on the ARM Cortex architecture [22]. However, these are used less frequently in drone applications.

4.3 Choice of zero-crossing detection method

The control algorithm does not need the BEMF (Back Electromotive Force) voltage readings itself. Instead, phase information should be extracted from the BEMF by detecting the zero crossings of the BEMF. There are two possible methods to detect zero crossings in the BEMF from a brushless motor. The phase voltages can either be

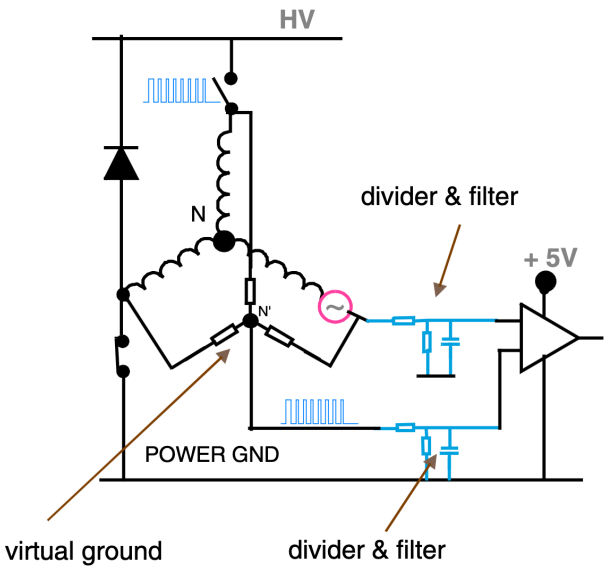


Figure 4.1: Analog comparator method [24]

compared with the neutral voltage using an analog comparator, or the phase voltages are integrated. This section does not go in depth into the mathematics or sampling and control theory, since that is covered in the control subgroup thesis [23].

4.3.1 Analog comparator

The STM32 microcontroller contains 4 analog comparators, which can be used to detect the sign of the BEMF with respect to the neutral voltage, as shown in Figure 4.1. The main features of this method are:

- requires few calculations, leaving the CPU free for other tasks.
- reacts very quickly, with an interrupt being triggered on a sign change.
- sensitive to noise, which can cause false triggers or requires a large hysteresis to be configured. A large hysteresis requires a large amplitude of the BEMF, meaning that it does not work at low motor speeds.
- requires more external components (resistors and capacitors) and it requires six analog pins on the microcontroller for the three comparators.

During testing, it is noticed that the false trigger rate is extremely large. When the input signal has an amplitude of less than 200 mV, which is equivalent to 15% throttle, it is impossible to discern zero crossings from external interference. This also happens when the largest possible hysteresis is configured in the comparator.

4.3.2 Digital integration

If samples of the analog signal are taken at a high enough frequency, the samples can be added to perform digital integration. Figure 4.2 shows a simulation of how the zero crossing can be found by searching for the maximum of the integral. In Figure 4.2, random noise is introduced with a range of $(-0.25, 0.25)$. The main features of this method are:

- It is very resilient to interference and does not produce multiple triggers when more than one zero crossing occurs due to noise.
- It requires some calculations, consisting mainly of additions.
- Fewer external components are needed compared to an analog comparator.
- Only three analog pins on the microcontroller are needed for sensing the three phases.

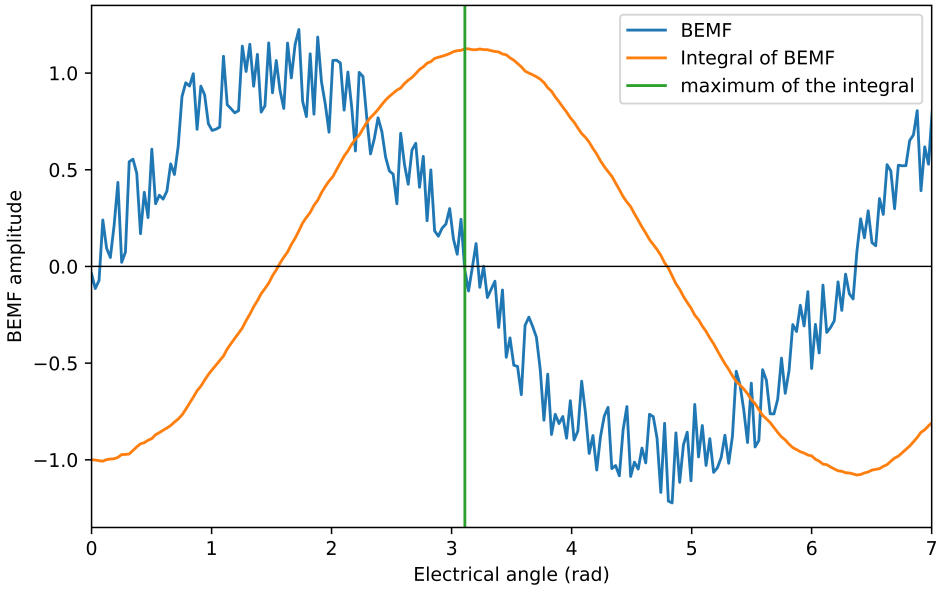


Figure 4.2: Using digital integration, the maximum of the integral of the BEMF is found. This simulation shows that this method is very resilient to noise or interference.

Although this method has not yet been tested, the performance is expected to be significantly better than the comparator-based method because of its resilience to noise. This resilience is caused by the averaging behavior of integration. The processing cost only equates to approximately ten clock cycles every 10 µs when samples are processed at 100kHz, which synchronizes them with the maximal switching speed. This is a CPU usage of 0.6% when the STM32 operates at its maximal frequency of 170MHz, which is why this method is chosen to be implemented if voltage sensing is used by the control subgroup.

Chapter 5

PCB design

5.1 First Prototype

Because it is easy to make a mistake during PCB design, and because we need a development board to test our code on, it is decided to design a small prototype PCB that is separate from the inverter. This PCB contains many features about which it is not yet decided how they are best implemented. Figure 5.1 shows the prototype design, with an interface header on the right side of the board. This header is meant to connect the PCB to the inverter board designed by the hardware subgroup. Figure A.2 shows the schematic of the prototype PCB.

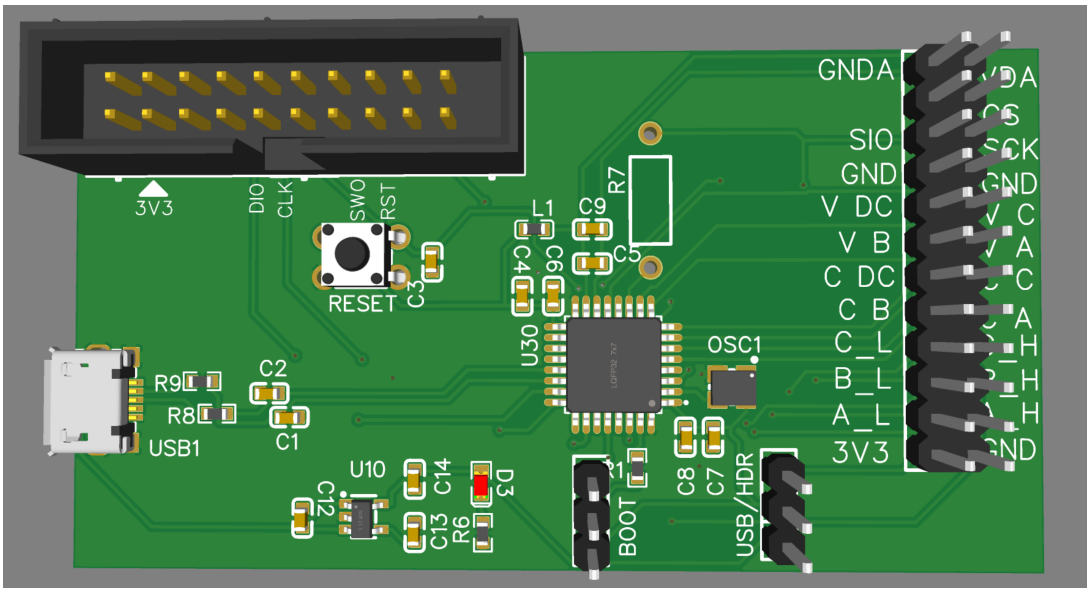


Figure 5.1: A render of the first Prototype PCB. It only has components on one side of the board

5.1.1 Summary of prototype PCB features

- The board has pins exposed for four different programming interfaces. However, in the end, only one was used. The SWD (Serial Wire Debug) interface uses two pins for power and two pins for a clock and data line. It can be used to program and debug the microcontroller [25].
- The PCB has an external oscillator crystal, which can offer a more precise clock than the internal oscillator of the microcontroller.
- The board separates digital and analog power supplies to reduce the interference on the analog signals caused by digital switching noise. Both have small decoupling capacitors close to the microcontroller, and larger capacitors further away. This follows the reference design (Figure A.1) from the *STM32G431 getting started guide [26].*

- *The pinout of the prototype can be found in appendix B. Initially, some functions were placed on arbitrary pins, but this was changed for the final prototype.*

5.1.2 Lessons learned from the first prototype

- Only the SWD interface is needed for programming and debugging.
- The external oscillator is not needed because the internal clock is precise enough for all timing-critical features such as the DShot link. Also, the clock trace causes interference on nearby analog signals.
- Separation of power supplies is crucial for low noise analog measurements.
- ground planes need to be placed below every signal to increase signal integrity. Signals should not cross other signals.
- the pin assignment of the microcontroller should be grouped per use. Analog and digital signals should be on separate sides of the microcontroller to aid in reaching the previous two points.

5.2 Final Prototype

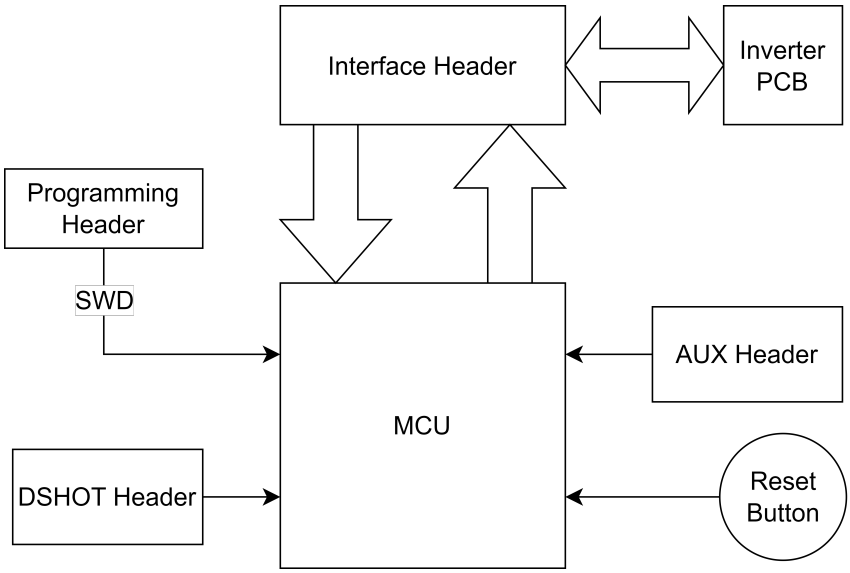
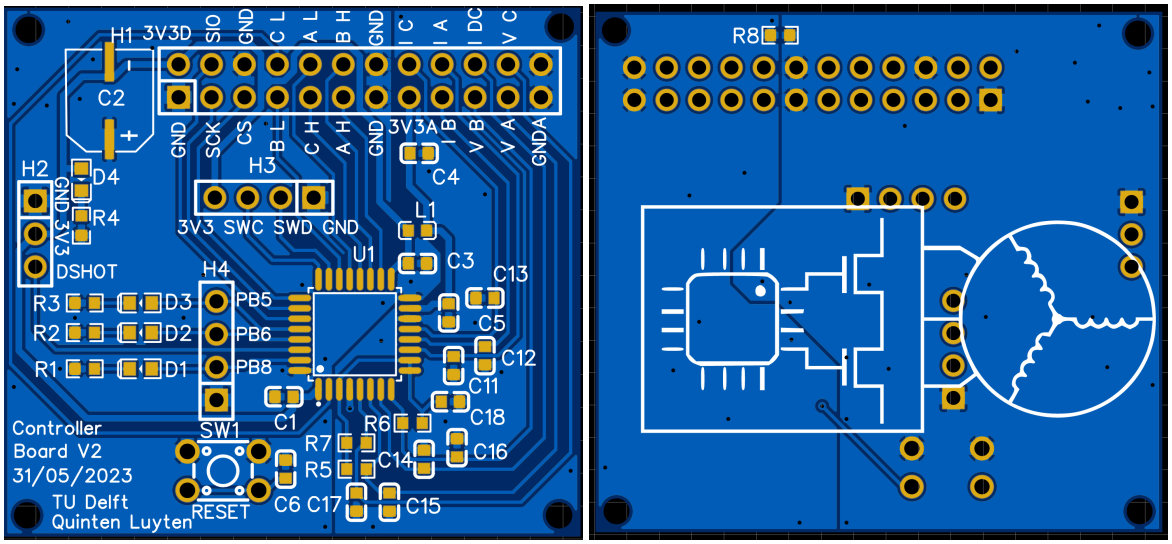


Figure 5.2: Block Diagram of the final prototype PCB

The final prototype is designed after the first prototype PCB was thoroughly tested. Certain redundant features are removed, and some pin functions are swapped to better align with the internal logic of the STM32. For example, the motor PWM output signals can now be connected to a single internal motor control timer. The schematic is still largely based on the reference design (Figure A.1), but now includes a separate DShot connection. Also, every analog signal is filtered before it is digitized. The full schematic is shown in Figure A.3. All digital interfaces are along the left side of the board. A separation line runs through the middle of the board, which isolates the analog and digital functionalities to reduce interference (Figure 5.3).

5.2.1 Power supplies

The final PCB is meant to receive two separate power supplies and keeps these electrically isolated from each other. The inductive bead L1 and solder bridge R8 can be used (Figure 5.3), if necessary, for testing with a single power supply. However, they are intended to remain open. Both the analog (3V3A) and digital (3V3D) voltages are supplied by the inverter board through the interface header at the top of the board and have separate grounds (GND and GNDA). The large capacitor C2 in the top left corner of the PCB has a capacitance of 220 μF (Figure A.3). This is the largest capacitor value that is easily found in a 5 mm SMD (surface mount device) format. Any larger footprint would take up a lot of space while providing little extra benefit. Smaller decoupling capacitors are placed close to the microcontroller as per the getting started guide [26].



(a) (b)

Figure 5.3: Final PCB. The separation line runs diagonally underneath the microcontroller and separates the digital and analog domains. Ground planes cover the entire back side of the PCB.

5.2.2 Digital interfaces

Along the left edge of the board, there is a header for the DShot link to the flight controller. The trace to the pin is kept as short as possible and completely above a ground plane to reduce noise.
The interface header is located along the top edge of the board. It still has the same functionality as in the prototype PCB, which is to connect to the inverter PCB. However, the pinout has been changed to better separate analog and digital signals. Below the interface header, there is a downsized SWD interface, as it became clear from testing that we did not need any other pins for programming and debugging. The USB port has been removed since its pinout interfered with the motor control pins and it does not provide any extra functionality. In its place, a small 4-pin header has been placed that makes the unused pins accessible in case they are needed for debugging purposes. To each unused pin, an LED has been attached for the same reason.

5.2.3 Analog interfaces

Since accurate measurements are needed for the control algorithm, capacitors are placed close to the analog interfaces to reduce high-frequency noise. Figure 5.3A shows these capacitors in the bottom right of the board, as close to the microcontroller as possible. All the capacitors C11 through C18 are decoupling to ground.

The analog circuitry is designed for two functionalities. These are BEMF sensing through voltage dividers, and current sensor readout. The first is used in control algorithms such as trapezoidal control, while the latter is used for FOC (Field Oriented Control) and other algorithms [23]. Both options are supported on the PCB because the control subgroup of the project did not yet decide which algorithm is best implemented when the design for the PCB was submitted.

Phase current measurement

The phase currents are measured by a Hall-effect current sensor on the inverter PCB [27]. Its analog output is proportional to the current flowing through nearby conductors. The analog output of the current sensor is driven by a low output impedance op amp, which means that a capacitor on the output does not significantly limit the bandwidth of the sensor. However, the bandwidth of the sensor is limited by the reaction time of the Hall element. The datasheet recommends a 10nF capacitor on the output for noise management [28], which is why $C_{11-14} = 10nF$.

BEMF measurement

The maximal frequency of the analog signals is 2.4 kHz (Section 2.3). To avoid too much of a delay in the BEMF measurement, the cutoff frequency is chosen to be 20 kHz. This is one order of magnitude higher than the maximal electrical frequency that the motor will operate at [24]. However, the cutoff frequency is low enough that it filters out most of the switching interference caused by the inverter. Equation 5.1 gives the value for these capacitors on the voltage measurement. The resistor value is 3kΩ, which is the resistance of the voltage divider implemented in the inverter PCB [27]. The closest standard value is 2.7 nF.

$$C_{filter} = C_{15-17} = \frac{1}{2\pi f R} = \frac{1}{2\pi \cdot 20kHz \cdot 3k\Omega} = 2.65nF \tag{5.1}$$

5.2.4 Effects of non-idealities

Although none of the signals on the PCB operate above a frequency of 1 MHz, it is still important to consider effects such as capacitive crosstalk, signal loops, and other possible noise sources. Also, the square waves of the DShot line and the gate drive signals introduce higher order harmonics which exceed 1 MHz.
Great care was taken to not disturb the continuous ground plane on the back of the PCB to prevent signal loops that can act as antennas. Since none of the signals on the PCB are differential pairs, all signals have their return signal flowing through one of the ground planes, and due to the almost undisturbed ground plane, it can be guaranteed that no unnecessary noise is induced because of signal loops. The only exception is the crossover between the trace connecting the "Reset" button to the microcontroller and the trace connecting R7 and R5 to the negative comparator inputs on the microcontroller. This was deemed acceptable since the reset line is a very low-speed connection that is not sensitive to interference. It was later decided that these resistors do not need to be populated, since the internal comparators connected to these resistors are not used (Section 4.3).

There are a lot of parallel traces on the board. The closest of these are $s = 0.5mm$ apart. Equation 5.2 gives an estimate of the capacitance between adjacent traces [29].

$$C(pf/cm) = 0.12\frac{t}{w} + 0.09(1+k)log_{10}(1 + 2\frac{w}{s} + \frac{w^2}{s^2}) \tag{5.2}$$

For a PCB with a dielectric constant $k = 4.2$, a trace thickness $t = 0.035mm$, and trace widths $w = 0.254mm$, this results in a capacitance of $0.24pF/cm$. With the longest parallel traces having a length of 54 mm, the maximal capacitance is $1.3pF$, which is negligible compared to the filtering capacitors C11 through C18 (Section 5.2.3).

Chapter 6

Programming of the microcontroller

ST Microelectronics offers an all-in-one IDE (Integrated Development Environment) for the STM32 series of microcontrollers. It allows configuring, writing code, programming, and debugging in one piece of software, called STM32CubeIDE. This was used during the entire project. Appendix C contains shortened versions of the program files for the project. The complete codebase can be found on the github page for the project: github.com/Quintenluyten/BAP-ESC-Releases [30]. As explained in Chapter 5, the microcontroller has an SWD interface that can be used with an ST-Link programmer to program and debug the device. This chapter covers all the blocks from the block diagram in the introduction (Figure 1.2).

6.1 DShot front end

On the STM32, a pin cannot be active as an input and an output pin simultaneously. That is why the pin configuration of the DShot pin needs to be changed after every packet of data.
When the pin is configured as an input pin, it is connected to timer 3 as an input capture pin. When the logic level on the pin changes, this generates an interrupt in the program and it captures the time since the last logic level change. The interrupt routine uses this time to measure the pulse length of an incoming signal, which can later be compared to a threshold time to determine whether a '1' bit was received or a '0' bit. The edge detection interrupt routine does not directly calculate the bit value to minimize the calculation time.
When the pin is configured as an output pin, it is connected to the GPIO (General Purpose Input/Output) B register. Timer 4 is configured to give a pulse every 1.33 µs. When this pulse occurs, a DMA (Direct Memory Access) request moves a value from the telemetry array to the GPIO B registers to set the pin high or low. The last entry of the telemetry array is always a '1' to set the DShot line back to its idle logic high level.

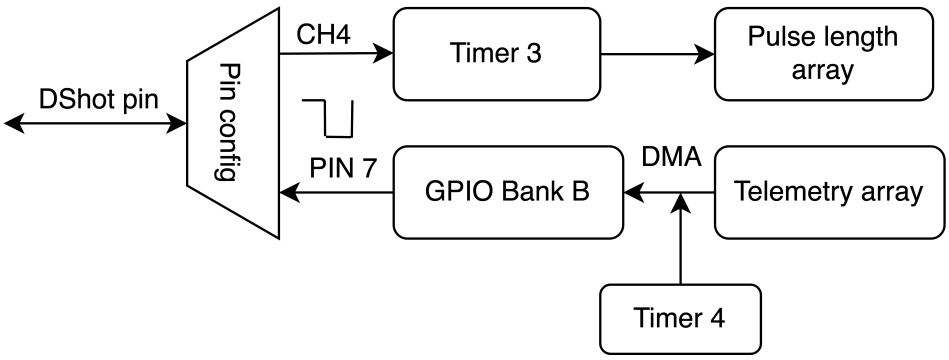


Figure 6.1: Overview of the DShot transmit-and-receive interface. The pulse length and telemetry arrays can be accessed from memory. The pin configuration is switched after every transmission and receive cycle.

The implementation of this algorithm in practice takes a lot of testing. Due to the high speed of the signal, care must be taken to guarantee the signal integrity. Also, if the interrupt routine is disturbed by other processes running on the microcontroller, it can produce incorrect pulse length measurements. And if the interrupt routine takes more than 420 ns to be executed, another edge in the signal can occur before the previous edge is processed (Table 3.1).

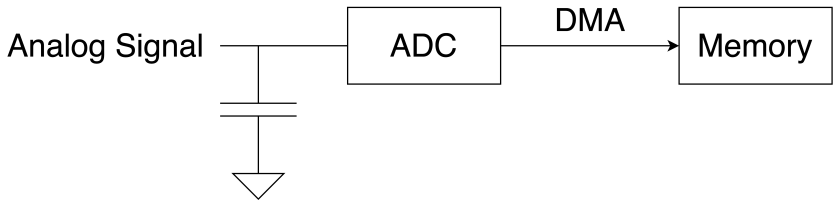


Figure 6.2: Data flow through the ADCs. The Direct Memory Access (DMA) routine automatically updates the digital value in memory.

6.2 DShot package processing

The packets that are received by the MCU are processed and interpreted in the following steps:

1. Confirm if the CRC is correct, if not then the package is discarded
2. Convert the value in the packet to a command or a throttle amount
3. Rescale the throttle with the factor from the temperature management system (Section 6.5)
4. Prepare and send telemetry to the flight-controller

To send back information to the flight controller, packages have to be prepared according to the specifications in Chapter 3. It is implemented by using the following steps:

1. Check if extended telemetry data or an eRPM package must be sent
   (a) If an eRPM package must be sent: get the value from the control algorithm and convert it to the exponential format.
   (b) For the extended telemetry: get the values from the sensor readout and append the correct prefix (Table 3.2. It cycles between temperature, voltage, current, and a status frame indicating the stress-level.
2. Add the CRC to the package
3. Encode the package with GCR
4. Apply differential encoding to the package
5. Set the package in the DMA registers
6. Enable the transmission

6.3 Analog sensor readout

As discussed in Section 4.3, the control algorithm can be supplied with phase voltage readings. However, if the algorithm requires phase currents instead, these can also be provided. The principle for both kinds of values is the same: the ADCs of the microcontroller continuously convert analog signals to digital values using a DMA routine, and when requested, the control algorithm can immediately read the most recent sample from memory. The advantage of this method compared to request-based sampling is that the sampling occurs in the background, and there is no need to wait for a sample to be taken. Also, the sample can be read exactly in the middle of the PWM (Pulse Width Modulation) on-period for maximal reduction of interference caused by the PWM switching events on the BEMF. The filtering and interpretation of the digitized phase currents and voltages is not further discussed here, as this is part of the control algorithm [23].

6.4 Temperature sensor readout

The Inverter subgroup decided to use the TMP127 digital temperature sensor for its large input sensing range and accuracy. The accuracy of a sensor is easier to guarantee on digital sensors since they are delivered already calibrated and the accuracy is independent of the routing or the conversion method used. The sensor uses half-duplex SPI, but if only a temperature value is needed, it can also be used in slave read-only SPI mode [31]. In this

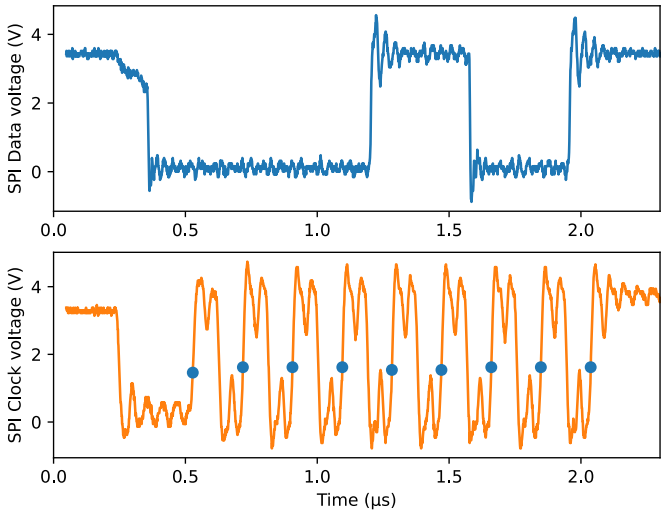


Figure 6.3: Temperature reading package. At every rising edge of the SPI Clock, indicated with blue dots, the SPI data line is read. Here, the value “000011001” is read, which is 25 °C.

mode, the data line of the sensor is connected to the MISO pin of the STM32. The STM32 acts as the SPI master and controls the SPI clock. When a temperature value is requested, the master activates the CS line and the clock. The temperature sensor outputs the temperature, MSB first, as visible in Figure 6.3. When the desired precision is received (up to 14 bits), the master turns off the clock and deactivates the CS line. In Figure 6.3, and in the code, only 9 bits are captured since this gives a precision of 1°C. The first bit is a sign bit that is ignored since the operating conditions are above freezing temperature. The 8 remaining bits give the temperature of the inverter in degrees Celsius. This one-byte value has a range from 0 to 255 degrees. The STM32 has full hardware support for SPI, so this process requires very little CPU usage and can run in the background.

6.5 Temperature managment

To comply with the requirement that the controller must actively prevent overheating (Section 2.4), a throttle limiter was implemented. When the temperature exceeds 80°C, the controller lowers the throttle sent to the control algorithm according to the curve shown in Figure 6.4. The exact shape of the curve is not very important, as long as the amount of heat being generated decreases as temperature increases. For this, the heat generated is assumed to be proportional to the square of the power delivered. This is because the loss increases quadratically with increasing current. The power limiter works by dividing the requested throttle by a factor that increases with temperature.

The controller also prioritizes sending temperature telemetry to the flight controller and sets the stress level to the maximum to indicate that overheating is occurring. This not only gives an indication to the flight controller and the pilot that the ESC is overheating, but it also prevents the flight controller from trying to compensate the loss in power by increasing the throttle. Without the indication, the motor speed control loop from the flight controller would fight the temperature management logic in the ESC.

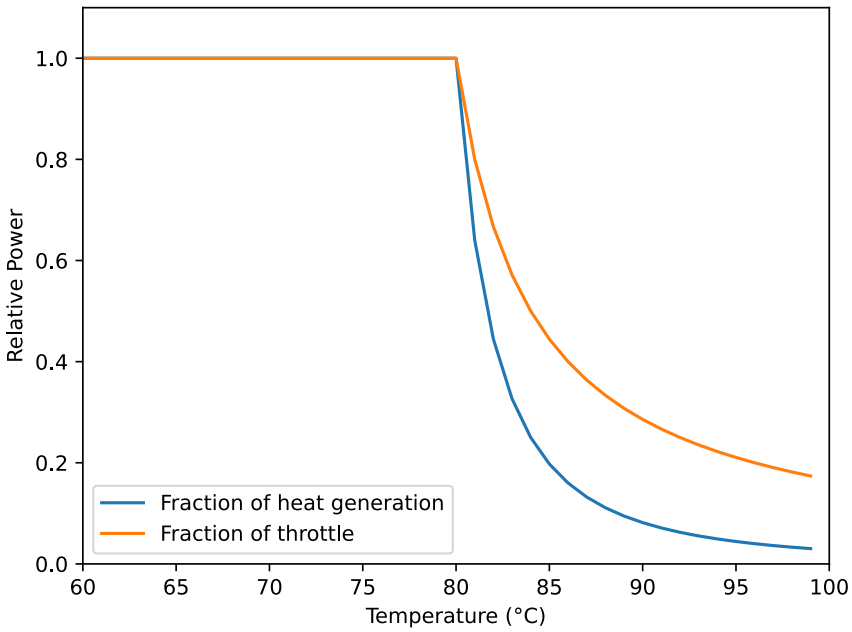


Figure 6.4: The throttle value which is forwarded to the control algorithm is a fraction of the value requested by the flight controller if the temperature of the inverter exceeds 80°C. The power loss in the inverter is thus also reduced to prevent overheating.

Chapter 7

Prototype testing and analysis

7.1 DShot Processing time

To compare DShot with other protocols, a good understanding of the properties of DShot is required. Since DShot does not have built-in hardware support on MCUs, it uses the CPU core of the MCU to process packets of data. To measure the processing time of this data, a GPIO output pin was turned on before the operation and turned off after the operation. The GPIO output pin was then measured on the oscilloscope to determine the length of execution. The results are shown in Table 7.1 and 7.2. The DShot implementation can be split into two parts. The first is the part that handles the bit detection, which uses interrupts. When 16 bits are detected, the package is then processed and a telemetry response is prepared. The CPU core is thereby occupied during a burst of smaller periods when a DShot packet is arriving, and during a longer period when the package is being processed.

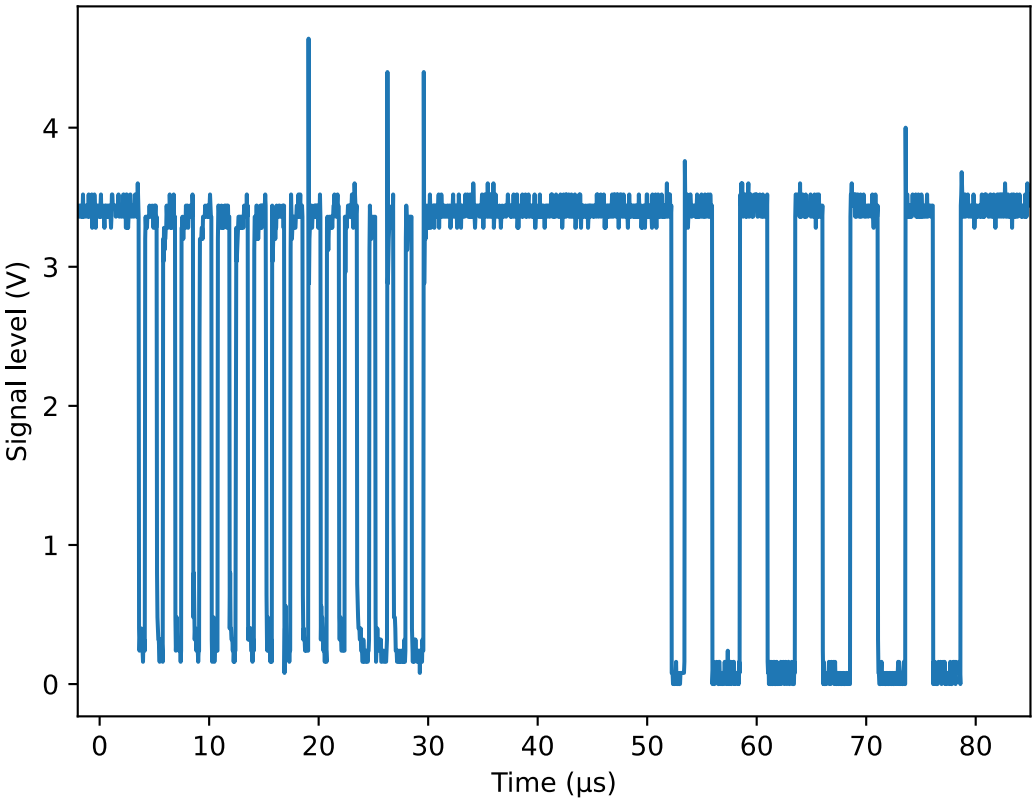


Figure 7.1: DShot package and response captured by an oscilloscope.

According to the protocol definition, the ESC needs to wait for 25 µs after the DShot throttle package is received, before the telemetry package can be transmitted (Figure 7.1).

Table 7.1: DShot time in CPU with a 2kHz package frequency

Section	Time	Clock Cycles
Packet Processor and telemetry	76 µs	13000
Wait between packet processing (Core is Free)	420 µs	71400
1 Edge interrupt	380 ns	65

Table 7.2: Partial Package Processing and telemetry time in CPU

Section	Time	Clock Cycles
Packet decoding	1.36 µs	231
CRC check	299 ns	50
Packet interpreter	327 ns	56
Telemetry preparation	5.4 µs	918
Delay 1 (Core can be interrupted)	21 µs	3570
Transmission start	3.1 µs	527
Delay 2 (Core can be interrupted)	44 µs	7480
Listening start	2.36 µs	401

The processes described in Tables 7.1 and 7.2 can be halted by an interrupt at any time, except during the edge detection interrupts. Section 6.1 explains the edge detection interrupt routine. None of the processes, except for the edge detection interrupts, are time-sensitive. The core is also available during delay periods, or when the DShot interface is waiting for the next package.
One edge interrupt takes 380 ns, and two interrupts happen for every bit, one for the falling and one for the rising edge (Section 6.1). Since the bit rate of DShot600 is 600kHz, a bit has a period of 1.67 µs. Equation 7.1 describes the percentage of time that the core is occupied when receiving data from the DShot line. As a result, the fraction of the time that the core is free equals 54%.

$$CPUfree(\%) = 100 \cdot \frac{BitTime - 2 \cdot t_{interupt}}{BitTime} = 54\% \tag{7.1}$$

7.2 DShot package corruption

Another measure of the quality of the DShot protocol is how many packages are lost in transmission. To test this a short connection to the flight controller was used, the flight controller was configured to only send the same data constantly. Knowing what data the MCU is supposed to receive is compared to the actually received data. Wrong data is then classified into two categories. The first is corrupted data that failed the CRC, and the second is corrupted data that passed the CRC. The result can be found in table 7.3.

Table 7.3: DShot package corruption measurements. The test lasted 5 mins (2kHz update rate) and the physical connection was kept relatively short to maintain signal integrity

Total packages	Corrupted Package	Incorrect CRC	Correct CRC
596899	20214	18882	1332
100%	3.386%	3.163%	0.223%

7.3 ADC Accuracy

As per the interoperability requirements (Section 2.7), the ADCs should have a linearity error of less than 0.25%. In order to verify whether the design satisfies this requirement, steady-state ADC measurements were made. Steady-state measurements are easier to produce and more accurate to verify with a multimeter, but do not guarantee that the accuracy will be maintained at higher frequencies of the input signal that has to be measured. However, the maximal frequency of the signals that has to be measured is less than 2.4 kHz (Section 2.3. The ADCs on the STM32 are able to operate at frequencies over 3 MHz [21]. With careful implementation of the design recommendations [32] as explained in Section 5.2.3, the accuracy should be maintainable, even at the maximal input frequency.
The current sensors on the inverter PCB produce their output relative to the analog power supply. The program

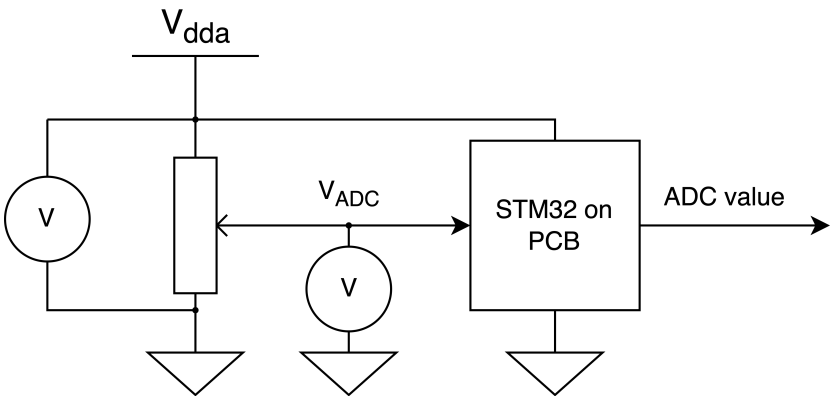


Figure 7.2: ADC accuracy measurement test setup, using a potentiometer to change the analog voltage.

V supply	V meas.	Count
3.329	0	0
3.332	0.06	80
3.331	0.28	340
3.329	0.49	593
3.331	0.76	911
3.332	0.87	1041
3.331	1.15	1366
3.332	1.32	1574
3.330	1.36	1619
3.333	1.56	1850
3.330	1.63	1935
3.333	1.85	2199
3.333	1.91	2275
3.331	2.15	2559
3.333	2.32	2770
3.332	2.44	2915
3.332	2.62	3144
3.332	2.87	3466
3.332	3.17	3895
3.329	3.32	4094

Table 7.4: ADC accuracy measurements. The ADC Count is a 12-bit variable with a range from 0 to 4095.

and PCB are built to also use the relative voltage reading compared to the analog power supply rail. The ADCs in the STM32 use the VDDA pin as the reference voltage for digitization. This ensures there is no dependency of the ADC output to the analog power supply voltage. The STM32 includes a self-calibration routine to remove errors and offsets from the measurements by using internal reference voltages [32]. This removes most non-idealities found inside the ADC, leaving only external noise sources to influence the measurements.
Figure 7.2 shows how the measurements were made. A potentiometer varies the analog voltage supplied to the ADC, with a multimeter measuring this voltage. Also, the VDDA measurement is made, which is necessary because the required reading is a relative voltage. During the processing of the data, the multimeter measurements are first normalized with respect to the VDDA voltage.
Table 7.4 shows the measurements, with $R^2 = 0.9975$ for $N = 20$ samples. The biggest deviation is 71 mV or 2.1%. This deviation is most pronounced in the range from 2.0 to 3.0 Volts, and is probably caused by integral non-linearity error (INL) in the ADC [32]. This is consistent across multiple tests.

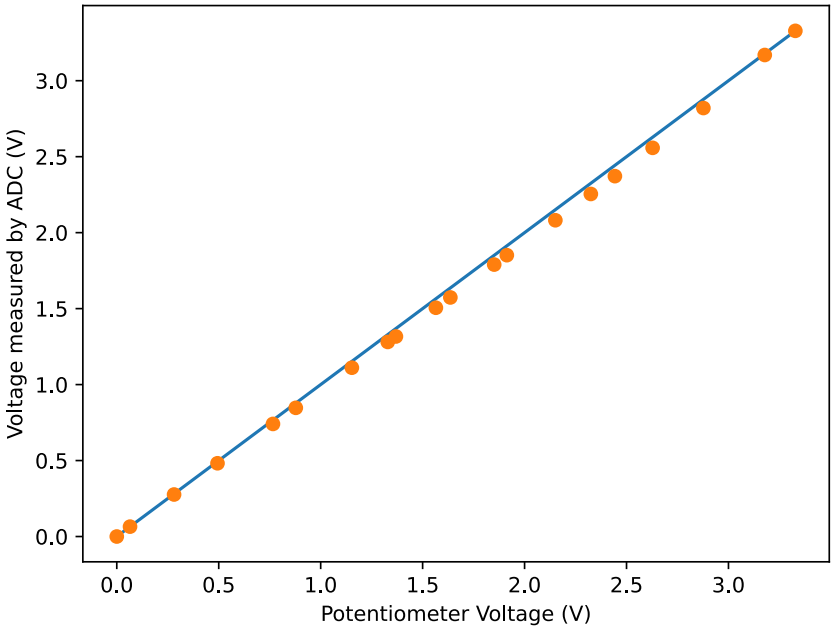


Figure 7.3: Ideal ADC transfer curve (blue) and actual measured points (orange)

Chapter 8

Conclusion, Discussion, and Outlook

8.1 Conclusion

The required features are implemented. The DShot interface is able to send and receive data. Also, the sensors can be read. This data is transmitted as extended DShot packets. Integration with the hardware for sensor readout is already tested to be working. However, integration of these features with the control algorithm can uncover errors in the code that have not been discovered so far. The control algorithm will directly control the gate signals through a peripheral timer on the microcontroller, but this functionality is not yet tested.
Table 8.1 shows the design requirements that the current design does not fulfill.

Table 8.1: Failed design requirements

	Requirement	Explanation
1.	The ESC must be able to satisfy the telemetry requirements when 90% of DShot packages are corrupted in transmission.	This is not possible within the DShot600 specification, since the maximal update rate of 8kHz does not allow a package to be received every 1 ms when 90% of packages are corrupted.
2.	The communication interface should forward any anomalies detected by the control algorithm to the flight controller using DShot telemetry	This could not be tested without the integration of all modules of the project.
3.	The ESC must be able to restart without intervention after power interrupts of any duration.	This could not be tested without the integration of all modules of the project. However, the communication interface is automatically restarted after a loss of power.
4.	The ESC must be able to run continuously at its rated supply current and voltage for at least 30 minutes.	This could not be tested without the integration of all modules of the project. However, the controller has no problem operating for 30 minutes on its own.
5.	The digital values must not have more than 0.25% full range output linearity error with respect to the best fit straight line over the entire signal range at room temperature.	The ADC inside the microcontroller has a deviation of up to 2% of the full range.

The first requirement is not achievable with the DShot600 protocol. However, with the measured corruption rate of 3.4% and an update rate of 2 kHz, the telemetry requirements are met.

The testing also showed that the ADCs could satisfy the accuracy requirement, except for the range from 2 to 3 Volts input signals, where the maximal non-linearity is 2.1%.

8.2 Discussion

8.2.1 Implementation of DShot

As discussed in Chapter 7, DShot has some disadvantages. The first of which is that there are a lot of packages being rejected because they are corrupted in transmission. The DShot CRC is not an error-correcting check, but only an error-detecting check. As shown in Table 7.3, around 3% of the packages sent from the flight controller are not received by the ESC, even in a low-noise environment with short cabling.

Another issue with the DShot protocol is its low-quality CRC, as shown in Table 7.3. 0.2% of packages are still accepted by the CRC algorithm, even when they have the wrong value. The CRC algorithm only detects 93% of corrupted packages. This is because the CRC is not able to protect against data shifting. If the data is shifted to the right or left and the dropped bit is collected on the other side, the packet still has a valid CRC. However, even a perfect 4 bit CRC algorithm still has a $\frac{1}{2^4} = 6.25\%$ probability of giving a correct CRC for a corrupted package, purely by chance. Packages with corrupted data can have serious effects on the motor. If zero throttle is constantly sent by the flight controller to the ESC and a shift corruption occurs, then the throttle value read by the ESC can be equal to half the total throttle, causing serious damage. A fix for the shift corruption is to have the bits time out if they take too long. This would however require extra clock cycles on the CPU that are limited.
The protocol also uses a lot of CPU time, since the data reception and transmission needs to be handled by the CPU core. The STM32 microcontroller contains certain peripherals, such as timers and DMA channels. These take much of the load off the CPU, but are not as autonomous as a communication peripheral, as those available for SPI or I2C.
Also, the documentation and precise definition of the communication standard is vague and sometimes even in conflict with common implementations. The question can be raised why one would use DShot for communication between ESCs and flight controllers. One argument is that digital communication protocols allow error checking and can be more resilient to EMI than analog protocols (Section 1.4.1), but this also applies to CAN, SPI, or I2C.

8.2.2 Alternative Communication protocols

However, many standardized digital communication protocols already exist. Many of these are designed for communication between microcontrollers and work with a single wire. For example, single-wire USART is available on the STM32. Alternatively, the CAN bus protocol is used very often in vehicles and aviation and has a bit rate of up to 1 Mbit/s for the standard high-speed CAN protocol [33]. The STM32 also supports SPI, which is already used for the temperature sensor readout. The SPI interface on the STM32 has been tested to work flawlessly up to 5Mbit/s, which could allow four or six motors to be connected to one SPI bus, while still offering a larger bandwidth than the DShot600 protocol, and reducing the CPU usage. The only drawback would be the increase in wiring, as a clock line, two data lines, and multiple chip-select lines would be necessary.

8.2.3 Choice of controller

During the design of the communication, it became clear that with many processes running in parallel, a controller that is able to execute more tasks in parallel could perform more efficiently. The current MCU is still able to execute all the tasks with the help of interrupts and hardware modules, but some of its capabilities like the clock frequency could be reduced if more was done in parallel. If this optimization allows for a lower performance microcontroller to be used, it can reduce the cost of the ESC.

8.2.4 Measurement accuracy

Although the maximal deviation of the ADCs exceeds the accuracy requirement set in Section 2.7, the maximal error of 2.1% is not expected to give problems during integration if the input signals use the full range of the ADC. However, if a signal that deviates between 1 Volt and 2.3 Volts, such as a bidirectional current sensor with lower than ideal sensitivity, is used as an input signal, the error becomes larger with respect to the full range of the input signal. The maximal error of 71 mV is then 5.5% of the full range of the input signal, which could influence the controllability of the ESC.

8.3 Recommendations and Future Work

The design can still be improved in various ways. The current implementation of DShot uses a lot of time in the core. It can be investigated if more of the hardware modules can be used to run DShot and make it less core re-

liant. Another way to reduce time in the core is to switch to a different communication algorithm that has suitable built-in hardware modules. For example, most MCUs have hardware support for SPI.

It would also be valuable to design the ESC around a different controller type, like an FPGA. The challenges that arise during FPGA design, such as logic size limitations, ask for different solutions than those that exist for a microcontroller-based design, where processing time is the limiting factor.

With the current division of the project, the inverter hardware is completely separate from the microcontroller. ST Microelectronics offers the STSPIN32G4 motor controller with embedded MCU [34], which integrates the microcontroller with the power supplies and gate drivers which are currently placed on the inverter PCB. Work can also be done to merge the inverter PCB and the communication PCB, making the complete ESC more compact and easier to use.

Appendices

Appendix A

PCB design

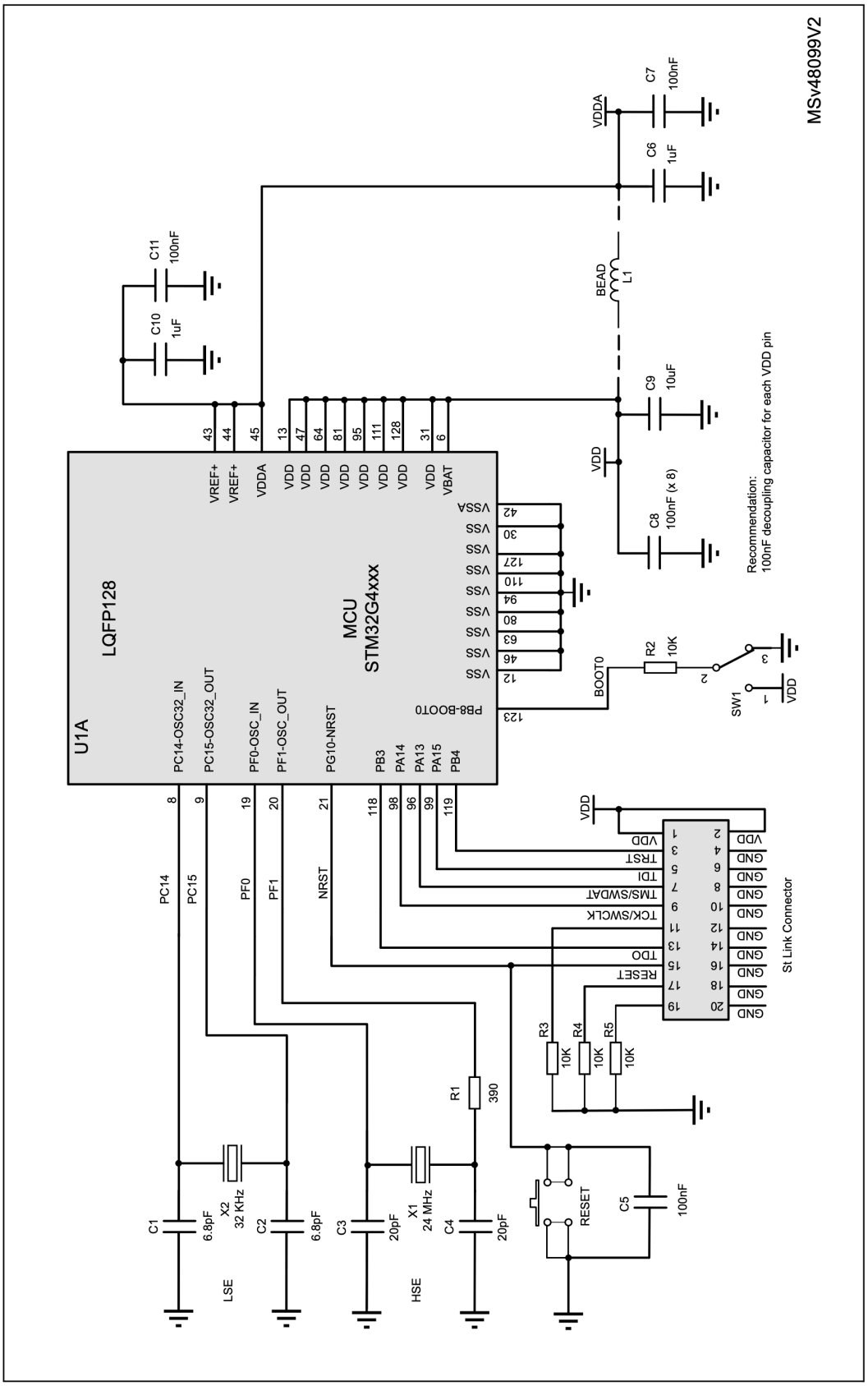


Figure A.1: Reference design for the STM32G4 microcontroller, from the STM32G4 series getting started guide [26]

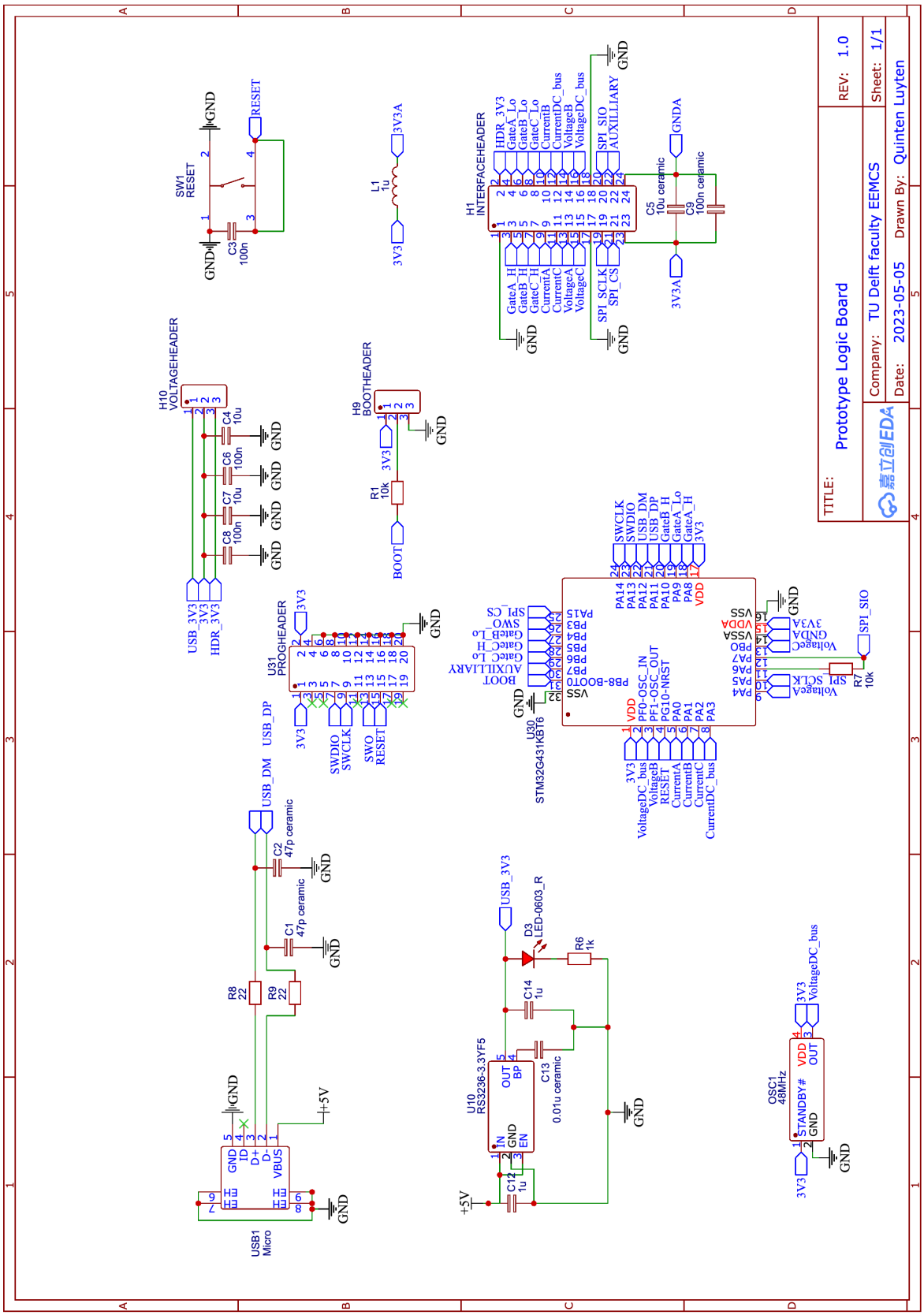


Figure A.2: First Prototype Schematic

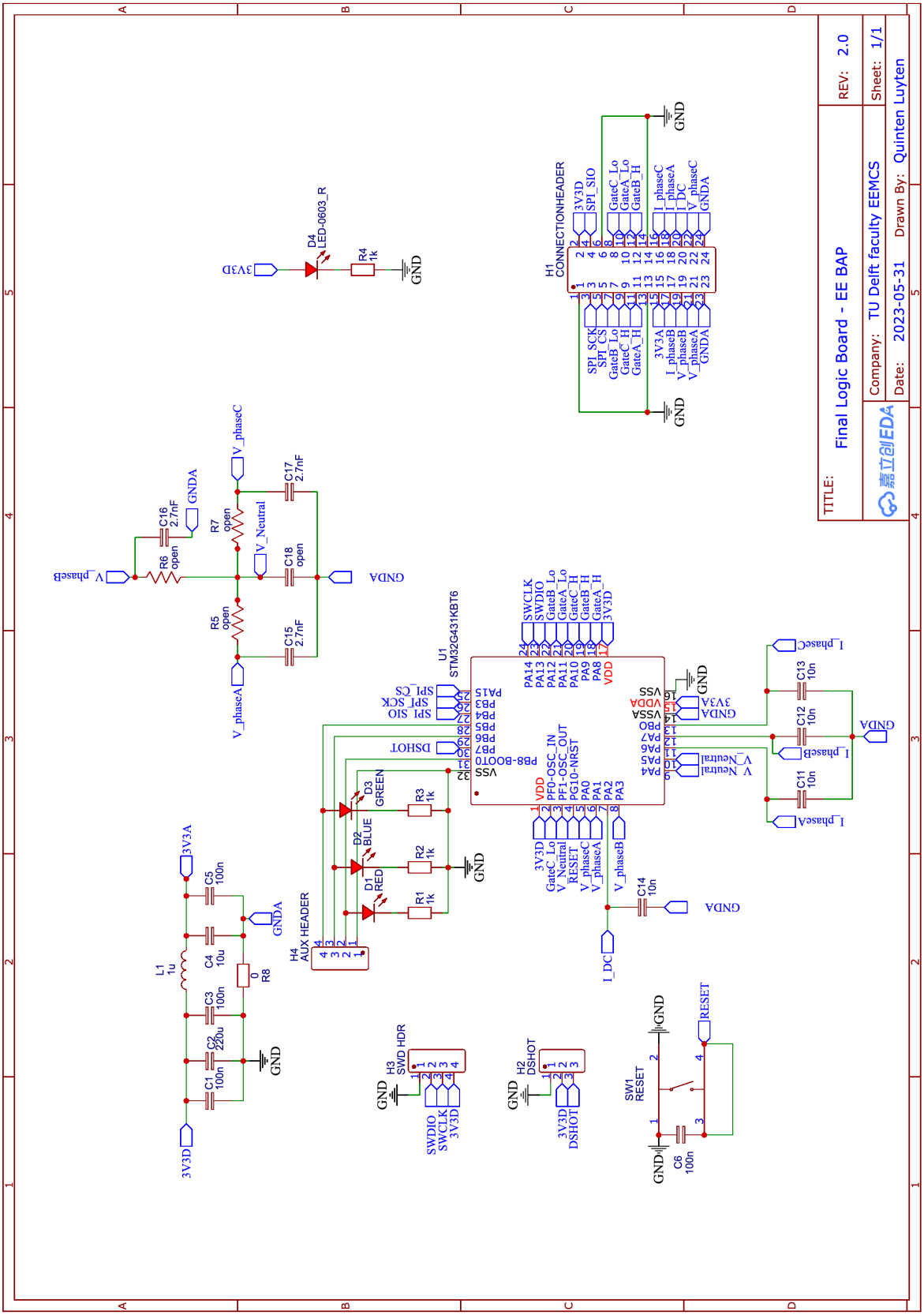


Figure A.3: Final Logic Board Schematic

Appendix B

MCU Prototype mapping

Table B.1: Mapping of the MCU on the prototype PCBs

Package pin number (LQFP32 package)	**Pin name**	**Mapping on the first prototype**	**Mapping on the final design**	**STM32 peripheral used on the final design**
1 and 17	VDD	Digital Supply Voltage	Digital Supply Voltage	VDD
2	PF0	Voltage DC bus	Gate C Low	TIM1 CH3 neg. output
3	PF1	Voltage B	Neutral voltage	COMP3 neg. input
4	PG10	Reset	Reset	NRST
5	PA0	phase A Current	phase C voltage	COMP3 pos. input
6	PA1	phase B Current	phase A voltage	COMP1 pos. input
7	PA2	phase C Current	DC current	ADC1 CH3
8	PA3	DC Current	phase B voltage	COMP2 pos. input
9	PA4	phase A Voltage	Neutral Voltage	COMP1 neg. input
10	PA5	SPI_SCK	Neutral Voltage	COMP2 neg. input
11	PA6	SPI_MISO	phase A current	ADC2 CH3
12	PA7	SPI_MOSI	phase B current	ADC2 CH4
13	PB0	Voltage C	phase C current	ADC1 CH15
14	VSSA	Analog Ground	Analog Ground	VSSA
15	VDDA	Analog Supply Voltage	Analog Supply Voltage	VDDA
16 and 32	VSS	Digital Ground	Digital Ground	VSS
18	PA8	Gate A High	Gate A High	TIM1 CH1 pos. output
19	PA9	Gate A Low	Gate B High	TIM1 CH2 pos. output
20	PA10	Gate B High	Gate C High	TIM1 CH3 pos. output
21	PA11	USB_DM	Gate A Low	TIM1 CH1 neg. output
22	PA12	USB_DP	Gate B Low	TIM1 CH2 neg. output
23	PA13	SWDIO	SWDIO	SWDIO
24	PA14	SWCLK	SWCLK	SWCLK
25	PA15	SPI CS	SPI CS	SPI CS
26	PB3	SWO	SPI SCK	SPI SCK
27	PB4	Gate B Low	SPI SIO	SPI MISO
28	PB5	Gate C High	Debug led 3	GPIO B
29	PB6	Gate C Low	Debug led 2	GPIO B
30	PB7	DShot	DShot	TIM3 CH4 - GPIO B
31	PB8/BOOT0	boot	Debug led 1	GPIO B

Appendix C

C code

The standard file formatting created by the STM32CubeIDE causes files with lots of empty or sparsely populated lines. This is why only the most relevant lines of code from the most relevant files are included in this appendix. This will not compile to a useable binary. The full code can be found on the Release GitHub: github.com/Quintenluyten/BAP-ESC-Releases [30]

C.1 Shortened main.c file

```
/* USER CODE BEGIN Header */
/* Written by Ruben Vos and Quinten Luyten
/* TU Delft BSc Electrical Engineering Graduation project 2023
/* USER CODE END Header */

#include "main.h"
#include "adc.h"
#include "dma.h"
#include "spi.h"
#include "tim.h"
#include "gpio.h"
#include "DSHOT.h"

#define ADCSIZE 64
#define TelemSize 21
#define MaxTemp 80

volatile uint8_t armed = 0; //Can there be signals to the gate drivers
volatile uint8_t EDT_Enabled = 0; // Is the Extended dshot enabled?
volatile uint32_t countervalue;
uint16_t lastreceivedDShotpacket[16] = {0};
uint8_t DShotbitcount = 0;
uint16_t dshotpacket = 0;
uint16_t throttle = 0;

volatile uint16_t MeasuredERPM = 0; //Measured ERPM from control group
uint16_t ADC1_BUFFER[ADCSIZE];
uint16_t ADC2_BUFFER[ADCSIZE];
uint8_t TempScalar=4;

uint32_t TelemetryArray[TelemSize] = {0,1,1,0,0,1,1,0,0,1,1,0,1,0,1,1,0,0,0,1,0};

volatile uint32_t CrossCounter=0;
uint8_t temp[1] = {0};
uint16_t temp_counter = 0;
```

```
uint8_t TelemetryCycler=0;
uint16_t ERPMCycler=0;
uint8_t StressLevel = 0;

void SystemClock_Config(void);


int main(void)
{
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_TIM3_Init();
  MX_TIM4_Init();
  MX_ADC2_Init();
  MX_ADC1_Init();
  MX_TIM1_Init();
  MX_SPI1_Init();

  /* Enable ADC DMA and disable the DMA Interrupt, which was enabled in MX_DMA_Init() */
  HAL_ADC_Start_DMA(&hadc1, (uint32_t*)ADC1_BUFFER, ADCSIZE);
  HAL_ADC_Start_DMA(&hadc2, (uint32_t*)ADC2_BUFFER, ADCSIZE);
  NVIC_DisableIRQ(DMA1_Channel2_IRQn);
  NVIC_DisableIRQ(DMA1_Channel3_IRQn);

  //Start Comparators
  HAL_COMP_Start(&hcomp1);
  HAL_COMP_Start(&hcomp2);
  HAL_COMP_Start(&hcomp3);

  /* Start TIM3 for Input capture */
  HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_4);
  HAL_TIM_Base_Start(&htim3);

  /* Start TIM4 for output DMA generation */
  HAL_TIM_Base_Start(&htim4);
  TIM4 -> DIER |= (1 << 8); // Update DMA request Enable (UDE)
  DMA1_Channel1->CCR |= (1<<1); //Enable Transfer Complete Interrupt
  DMA1_Channel1 -> CCR |= (1 << 5); // Circular mode
  HAL_TIM_OC_Start(&htim4, TIM_CHANNEL_1);
  HAL_DMA_Start(&hdma_tim4_up, (uint32_t) TelemetryArray, (uint32_t)&(GPIOB->BSRR), 22);
  DMA1_Channel1->CCR &= ~ 1; //Disable DMA routine for now

  while (1)
  {

          if(DShotbitcount == 16){
                  dshotpacket = decodeDShot((uint16_t*)&lastreceivedDShotpacket);

                  if(CRC_Check(dshotpacket)){

                          PacketInterpreter(dshotpacket);
                          throttle = (throttle<<2)/TempScalar; // Temperature Management
                          Diff_GRC_telemetry_package((uint16_t) 200);
```

```
                        /* Delay */
                        for(int i = 0; i < 450; i++){
                                volatile int a = i*i;
                        }

                        Start_transmission();

                         /* Delay before DShotbitcount can be set to 0 */
                         for(int i = 0; i < 1000; i++){
                                volatile int a = i*i;
                         }

                 }
                DShotbitcount = 0;

                /* Temperature sensor readout */
                if(temp_counter == 100){
                        temp_counter = 0;
                        HAL_SPI_Receive(&hspi1, temp, 1, 1000);
                        if(temp[0]>MaxTemp){
                                TempScalar = temp[0] - MaxTemp + 4;
                                TelemetryCycler = 0; //Send Temperature immediately
                                StressLevel = 15;
                        }
                        else{
                                StressLevel = 0;
                                TempScalar = 4;
                        }

                }
                else{
                        temp_counter++;
                }
          }
  }
}


void TIM3_IRQHandler(void)
{
        if ((TIM3->DIER >> 4) && (TIM3->SR >> 4)){
        // If the interrupt is caused by TIM3 capture compare event:
        // DIER >> 4: CC4 Interrupt enabled      SR >> 4: CC4 Flag set

                if(!(READ_REG(DSHOT_GPIO_Port->IDR)>>7)){
                        WRITE_REG(TIM3->CNT, 0);
                        SET_BIT(TIM3->CR1, 1);
                }
                else if (DShotbitcount < 16){
                        lastreceivedDShotpacket[DShotbitcount] = READ_REG(TIM3->CCR4);
                        DShotbitcount++;
                }
                TIM3->SR = ~(TIM_IT_CC4); //clear the flag
        }
}
```

C.2 DShot.c

```
/* USER CODE BEGIN Header */
/* Written by Ruben Vos and Quinten Luyten
/* TU Delft BSc Electrical Engineering Graduation project 2023
/* USER CODE END Header */

#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#include "gpio.h"
#include "main.h"


#define ADCSIZE 64
#define TelemSize 22

extern volatile uint8_t armed; //Can there be signals to the gate drivers
extern volatile uint8_t EDT_Enabled; // Is the Extended dshot enabled?
extern volatile uint32_t countervalue;
extern uint16_t throttle;
extern uint16_t MeasuredERPM; //Measured ERPM from controlgroups
extern uint32_t TelemetryArray[];
extern uint16_t ADC1_BUFFER[];
extern uint16_t ADC2_BUFFER[];
extern uint8_t temp[1];
volatile  uint32_t step3;
volatile  uint32_t step2;
volatile  uint32_t step1;
volatile  uint32_t step0;
volatile uint32_t test1;
volatile uint32_t Corrupted_Packets=0;
volatile uint32_t Total_Packets=0;
volatile uint32_t Check_Fail=0;



bool CRC_Check(uint16_t data)
{
        // Performs CRC check on incoming bidirectional DSHOT throttle packages
        if((data & 0x0F) == (~(data>>4 ^ (data >> 8) ^ (data >> 12)) & 0x0F)){
                return true;
        }
        return false;
}

//This function was stolen from the Betaflight open github
uint16_t CRC_Calc(uint16_t packet)
{
    // compute checksum
        uint16_t csum = 0;
        uint16_t csum_data = packet;
    for (int i = 0; i < 3; i++) {
        csum ^=  csum_data;   // xor data by nibbles
        csum_data >>= 4;
    }
    // append checksum
    csum = ~csum; //No inversion because telemetry is uninverted
```

```
    csum &= 0xf;
    packet = (packet << 4) | csum;

    return packet;
}

uint32_t GRC_encoder(uint16_t packet)
{
    uint8_t chunk;
    uint32_t GRC_encoded=0;
    uint8_t nibble;
    for(int i=0;i<4;i++){
        chunk = (packet >> (i*4)) & 0x0F;
        switch(chunk){
            case 0x0: nibble = 0x19; break;
            case 0x1: nibble = 0x1B; break;
            case 0x2: nibble = 0x12; break;
            case 0x3: nibble = 0x13; break;
            case 0x4: nibble = 0x1D; break;
            case 0x5: nibble = 0x15; break;
            case 0x6: nibble = 0x16; break;
            case 0x7: nibble = 0x17; break;
            case 0x8: nibble = 0x1A; break;
            case 0x9: nibble = 0x09; break;
            case 0xA: nibble = 0x0A; break;
            case 0xB: nibble = 0x0B; break;
            case 0xC: nibble = 0x1E; break;
            case 0xD: nibble = 0x0D; break;
            case 0xE: nibble = 0x0E; break;
            case 0xF: nibble = 0x0F; break;
            default:
                nibble = 1;
                break;
        }
        GRC_encoded |= nibble << (i*5);
    }
    return GRC_encoded;
}

uint32_t Diff_encoder(uint32_t packet){
        uint32_t Diff_encoded = 0;
        for(int i=20;i>0;i--){
                int lastnewbit = Diff_encoded&0x01;
                int packetbit = (packet>>(i-1))&0x01;
                Diff_encoded = Diff_encoded<<1;
                if(packetbit^lastnewbit){
                        Diff_encoded++;
                }

        }
        return Diff_encoded;
}

void SetPackageInRegisters(uint32_t packet){
        for(int i=0;i<TelemSize-2;i++){
                //if(i>21){
                //      TelemetryArray[TelemSize-1-i] = 0x800000;//0
                //}
```

```
                if((packet>>i) & 1){
                        TelemetryArray[TelemSize-2-i] = 0x80;//1
                }
                else{
                        TelemetryArray[TelemSize-2-i] = 0x800000;//0
                }
        }
        TelemetryArray[0] =  0x800000;//0

        TelemetryArray[21] = 0x80;//1
        //0x800000 = 0
        return;
}


extern uint8_t TelemetryCycler;
extern uint16_t ERPMCycler;
extern uint8_t StressLevel;

void Diff_GRC_telemetry_package(uint16_t period_us)
{
    uint32_t Diff_GCR_packet;

    if (ERPMCycler<1000){ // Calculate an eRPM telemetry package
        ERPMCycler++;
        if (period_us == 0 || period_us > 0x7FC0){
            return;
        }
        //Make it Exponentional
        uint8_t exponent = 0;
        while(period_us > 511){
            period_us = period_us >> 1;
            exponent += 1;
        }
        Diff_GCR_packet = (exponent << 9 | period_us); // this is the 12-bit data that
    }
    else {// Calculate an extended telemetry type
        ERPMCycler=0;
        uint8_t code;
        uint8_t value;
        switch(TelemetryCycler){
                case 0://Temperature
                        TelemetryCycler++;
                        code = 0b0010;
                        //TODO READ TEMPERATURE
                        value = temp[0];
                        break;
                case 1://Voltage
                        TelemetryCycler++;
                        code = 0b0100;
                        //TODO READ VOLTAGE
                        value = 178;// 44.5V
                        break;
                case 2://Current
                        TelemetryCycler++;
                        code = 0b0110;
                        value = ADC1_BUFFER[0]; //120 A
```

```
                        //TODO RESCALE VALUE WITH CORRECT FACTOR
                        value *= 1;
                        break;
                case 3:
                        TelemetryCycler=0;
                                code = 0b1110;
                                value = StressLevel<<1; //Nothing wrong
                                break;
                default://Weird ERROR?!?!?!
                        return;
                        break;
        }
        Diff_GCR_packet = (code << 8) | value;
    }
//     if(ERPMCycler < 8096){
//       Diff_GCR_packet = ERPMCycler;
//       ERPMCycler++;
//     }
    //Add the CRC
    Diff_GCR_packet = CRC_Calc(Diff_GCR_packet);
    //GRC encoding
    Diff_GCR_packet = GRC_encoder(Diff_GCR_packet);

    //Diff encoding
    //Diff_GCR_packet = 699050;
    Diff_GCR_packet = Diff_encoder(Diff_GCR_packet);

    SetPackageInRegisters(Diff_GCR_packet);
    return;
}

uint16_t decodeDShot(uint16_t *array){
        uint16_t packet = 0;
        for (int i = 0; i<16; i++){
                if(array[i]>80){
                        packet = (packet << 1) | 1;
                }
                else{
                        packet = packet << 1;
                }
        }
        return packet;
}

uint8_t ExtendedTelemToggleCount=0;

void PacketInterpreter(uint16_t packet){
        uint16_t value = packet>>5;

        switch(value){
                case 0: armed = 0; throttle=0; ExtendedTelemToggleCount=0; break; // Mo

                case 13: //Enabling extended telemetry
                        ExtendedTelemToggleCount++;
                        if(ExtendedTelemToggleCount==6){
                                EDT_Enabled = 1;
                                //TODOO :: SENT ANSWER MESSAGE
                                ERPMCycler = 1000;
```

```
                                TelemetryCycler = 3;
                        }
                        break;
                case 14: // Disabling extended telemetry
                        ExtendedTelemToggleCount++;
                        if(ExtendedTelemToggleCount==6){
                                EDT_Enabled = 0;
                                //TODOOO: SEN ANSWER MESSAGE

                        }
                        break;
                default:
                        ExtendedTelemToggleCount=0;
                        if(value>47){ //Normal throttle value
                                armed = 1;
                                throttle = value - 47;
                        }
                        break;
        }
}
```

C.3 Shortened tim.c file

```
/* USER CODE BEGIN Header */
/* Written by Ruben Vos and Quinten Luyten
/* TU Delft BSc Electrical Engineering Graduation project 2023
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "tim.h"

TIM_HandleTypeDef htim3;


void SET_TO_IC(void){

  /* Sets GPIO pin B7 to Input Capture mode */
  SET_BIT(GPIOB->AFR[0], 0xa0000000);
  TIM_IC_InitTypeDef sConfigIC = {0};
  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_BOTHEDGE;
  sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
  sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
  sConfigIC.ICFilter = 0;
  if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_4) != HAL_OK)
  {
        Error_Handler();
  }
  SET_BIT(TIM3->CCER, 0xb000);
  SET_BIT(GPIOB->PUPDR, (1 << 14));
  SET_BIT(GPIOB->MODER, (1<<15));
  CLEAR_BIT(GPIOB->MODER, (1<<14));
}


void SET_TO_GPIO_OUT(void){

  /* Sets GPIO pin B7 to GPIO Out mode */
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  GPIO_InitStruct.Pin = GPIO_PIN_7;
```

```
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /**/
  __HAL_SYSCFG_FASTMODEPLUS_ENABLE(SYSCFG_FASTMODEPLUS_PB7);
  SET_BIT(GPIOB->ODR, (1 << 7));

}

void Start_transmission(void){
        /* This function sets PB7 to output mode and
         * transmits the contents of the TelemetryArray
         *
         * Leave at least 30 microseconds for transmission
         * to complete before executing start_listening
         */
        SET_TO_GPIO_OUT();
        WRITE_REG(TIM4->CNT, 0);

        /* Enable DMA routine: start sending bits */
        DMA1_Channel1->CCR |= 1;
}
```

C.4 Shortened stm32g4xx_it.c file

```
/* USER CODE BEGIN Header */
/* Written by Ruben Vos and Quinten Luyten
/* TU Delft BSc Electrical Engineering Graduation project 2023
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "stm32g4xx_it.h"
#include "tim.h"

#define enable_DMA_ADC_interrupt 0

extern DMA_HandleTypeDef hdma_adc1;
extern DMA_HandleTypeDef hdma_adc2;
extern COMP_HandleTypeDef hcomp1;
extern COMP_HandleTypeDef hcomp2;
extern COMP_HandleTypeDef hcomp3;
extern DMA_HandleTypeDef hdma_tim4_up;
extern TIM_HandleTypeDef htim3;

/* This function handles the telemetry transmission DMA channel */
void DMA1_Channel1_IRQHandler(void)
{

  if(READ_BIT(DMA1->ISR, (uint32_t)DMA_FLAG_TC1)){ //If Transfer Complete interrupt
          DMA1->IFCR |= 7; //Clear TCI, HTI, GI flag
          DMA1_Channel1->CCR &= ~1; //Disable DMA routine
          SET_TO_IC(); // Start listening

  }
}
```

Bibliography

[1] O. Liang, *Flight controller explained: The ultimate guide to understanding fpv drone control systems*, 2023. [Online]. Available: https://oscarliang.com/flight-controller-explained/.

[2] "Fusion engineering, experts in flight control technology." (2023), [Online]. Available: fusion.engineering.

[3] mikeller. "Dshot telemetry #7264." (2020), [Online]. Available: https://github.com/betaflight/betaflight/pull/7264.

[4] *The betaflight open source flight controller firmware project*, 2023. [Online]. Available: https://github.com/betaflight.

[5] S. Skaug. "Blheli for brushless esc firmware." (2023), [Online]. Available: https://github.com/bitdump/BLHeli.

[6] M. Rasmussen and D. Manajipet. "Digital esc firmware for controlling brushless motors in multirotors." (2022), [Online]. Available: https://github.com/mathiasvr/bluejay/tree/main.

[7] Y. Chen, B. Xie, and E. Mao, "Electric tractor motor drive control based on fpga," *IFAC-PapersOnLine*, vol. 49, no. 16, pp. 271–276, 2016, 5th IFAC Conference on Sensing, Control and Automation Technologies for Agriculture AGRICONTROL 2016, ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2016.10.050. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896316316135.

[8] L. Diao, J. Tang, P. C. Loh, S. Yin, L. Wang, and Z. Liu, "An efficient dsp–fpga-based implementation of hybrid pwm for electric rail traction induction motor control," *IEEE Transactions on Power Electronics*, vol. 33, no. 4, pp. 3276–3288, 2018. DOI: 10.1109/TPEL.2017.2707639.

[9] X. Liu, Y. Xie, H. Chen, and B. Li, "Implementation on fpga for cordic-based computation of arcsine and arccosine," in *IET International Radar Conference 2015*, 2015, pp. 1–4. DOI: 10.1049/cp.2015.1306.

[10] S.-H. Song, J.-W. Choi, and S.-K. Sul, "Current measurements in digitally controlled ac drives," *IEEE Industry Applications Magazine*, vol. 6, no. 4, pp. 51–62, 2000. DOI: 10.1109/2943.847916.

[11] D.-W. Chung and S.-K. Sul, "Analysis and compensation of current measurement error in vector-controlled ac motor drives," *IEEE Transactions on Industry Applications*, vol. 34, no. 2, pp. 340–345, 1998. DOI: 10.1109/28.663477.

[12] E. Persson, "A new approach to motor drive current measurement," in *4th IEEE International Conference on Power Electronics and Drive Systems. IEEE PEDS 2001 - Indonesia. Proceedings (Cat. No.01TH8594)*, vol. 1, 2001, 231–234 vol.1. DOI: 10.1109/PEDS.2001.975317.

[13] C. Landa. "Dshot - the missing handbook." (2021), [Online]. Available: https://brushlesswhoop.com/dshot-and-bidirectional-dshot.

[14] Chris and D. Mosquera. "Extended dshot telemetry (edt) specification." (2023), [Online]. Available: https://github.com/bird-sanctuary/extended-dshot-telemetry.

[15] M. ILASLAN and T. AKINCI, "Fpga based reprogrammable main circuit board and auxiliary circuit board design," *Journal of Engineering Science and Technology*, vol. 15, no. 6, pp. 3955–3970, 2020.

[16] Lattice Semiconductor, *Pcb layout recommendations for bga packages*, 2022. [Online]. Available: http://www.latticesemi.com/view_document?document_id=671.

[17] ST Microelectronics. "Stm32 32-bit arm-cortex mcus." (), [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html.

[18] A. Drumea and M. Pantazică, "Aspects of using low layer count pcbs for embedded systems with fpga devices in bga packages," in *2016 IEEE 22nd International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2016, pp. 74–77. DOI: 10.1109/SIITME.2016.7777247.

[19] Silabs. "8-bit microcontrollers." (2023), [Online]. Available: https://www.silabs.com/mcu/8-bit-microcontrollers.

[20] Mouser Nederland, *Texas instruments digitale signaal-processoren & controllers - dsp, dsc*, 2023. [Online]. Available: https://nl.mouser.com/c/semiconductors/integrated-circuits-ics/embedded-processors-controllers/digital-signal-processors-controllers-dsp-dsc/?m=Texas+Instruments&product=DSPs%5C&sort=pricing.

[21] ST Microelectronics, *Arm ® cortex ® -m4 32-bit mcu+fpu, 170 mhz /213 dmips, up to 128 kb flash, 32 kb sram, rich analog, math accelerator*, 2021. [Online]. Available: https://nl.mouser.com/datasheet/2/389/stm32g431c6-1600866.pdf.

[22] NXP. "General purpose microcontrollers (mcus)." (2023), [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus:GENERAL-PURPOSE-MCUS.

[23] Y. Brodskaya and M. E. van Schagen, *Control Algorithm for a High Power Electronic Speed Controller*. 2023.

[24] ST Microelectronics, *Sensorless bldc motor control and bemf sampling methods with st7mc*, 2007. [Online]. Available: https://www.st.com/resource/en/application_note/an1946-sensorless-bldc-motor-control-and-bemf-sampling-methods-with--st7mc-stmicroelectronics.pdf.

[25] T. Cedro, M. Kuzia, and A. Grzanka, "Libswd serial wire debug open framework for low-level embedded systems access," in *2012 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2012, pp. 615–620.

[26] ST Microelectronics, *Getting started with stm32g4 series hardware development boards*, 2019. [Online]. Available: https://www.st.com/resource/en/application_note/dm00442716-getting-started-with-stm32g4-series--hardware-development-boards-stmicroelectronics
pdf.

[27] E. D. B. De Galembert and M. Simonart, *Design of an Electronic Speed Controller*. 2023.

[28] Allegro microsystems, *Coreless, high precision, hall-effect current sensor ic with common-mode field rejection and high bandwidth (240 khz)*, Dec. 2021. [Online]. Available: https://nl.mouser.com/pdfDocs/ACS37612-Datasheet.pdf.

[29] V. Armijo et al., "Lanl report on the r&d program for cathode strip readout chambers for the phenix muon tracking system," 1999. [Online]. Available: https://p25ext.lanl.gov/phenix/muon/phnotes/PN125/node1.html.

[30] R. Vos, M. E. van Schagen, Y. Brodskaya, M. Simonart, Q. Luyten, and E. D. B. De Galembert. "Bap-esc-releases." (2023), [Online]. Available: https://github.com/Quintenluyten/BAP-ESC-Releases/tree/main/Firmware.

[31] Texas Instruments, *Tmp127-q1 automotive grade, 0.8 °c spi temperature sensor with 175 °c operation*, 2022. [Online]. Available: https://www.ti.com/lit/ds/symlink/tmp127-q1.pdf.

[32] ST Microelectronics, *How to get the best adc accuracy in stm32 microcontrollers*, 2022. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/group0/3f/4c/a4/82/bd/63/4e/92/CD00211314/files/CD00211314.pdf/jcr:content/translations/en.CD00211314.pdf.

[33] ISO Standard, *11898-1. road vehicles-controller area network (can)*, 2015.

[34] ST Microelectronics. "High performance 3-phase motor controller with embedded stm32g4 mcu." (2022), [Online]. Available: https://www.st.com/en/motor-drivers/stspin32g4.html.