IN4073
Embedded Real-Time Systems
Report on stabilising a quadcopter

Delft University of Technology
September 15, 2020

Group 8 Hajo Kleingeld (4247248)
Corniël Joosse (4249623)
Eric Camellini (4494164)

Abstract

Flying a quadcopter requires a highly skilled pilot. To make this task easier, a controller system has to be developed including communication with a PC and filtering of sensor data. The resulting demonstrator model of our team was not able to fly fully stabilised due to a not properly working controller. Although, it has a stable communication link with the PC, logging facilities, a working filter and all the safety requirements implemented.

1 Introduction

A quadcopter (QR) is an aerial vehicle with four rotors. By increasing the speed of the rotors it is possible to give the QR lift, and by setting the rotors to different relative speeds, it is possible to steer. The problem with such vehicles is that they are quite unstable, so that it requires skills to be able to fly such a machine. Besides, there are high frequency distortions which are very hard to correct manually, so there is a need for an embedded system which controls and stabilises the movements of the QR.

The quadcopter used in this project is equipped with a soft-core processor with corresponding compiler, an acceleration sensor and a gyroscope. The rest of the system has to be developed by the team, which includes:

- A PC program to send commands to the QR;
- A lightweight protocol for communication between PC and QR;
- A filter for the sensor data;
- A controller which stabilises the QR and controls the engines;

2 Task division

The following table lists all the tasks for the project. The implementation of the PC program is done by one person, and therefore one task.

Documentation	All
PC program	Eric
QR program	
Communication and protocol	Corniël
Filters	Hajo
Controllers	Corniël
Scheduler	Hajo
Calibration	Eric
Engine mapping	Hajo
Fixed point arithmetic	Corniël
Test mode	Corniël
Main state-machine	Hajo
Logger (sensors)	Hajo
Logger (status/telemetry)	Eric

3 System design

3.1 Architecture

See appendix A for diagram and interfaces (page 9). The diagram is divided in three parts. A PC part, a FPGA part, and the FPGA communicates to the sensors and engines via an interface board, which is the last part. The list of interfaces is quite extensive, because most modules work tightly together and need therefore several interfaces to set or get data and to run the main task of the module.

3.2 Protocol

An important property of the protocol is that it has to be very lightweight. To be able to have a stable communication, a message should at least contain the following three items:

- A message start identifier to determine the start of a message
- Identification of the payload content and length
- Some sort of error check

The protocol is designed in such a way that for each of the three items we only need one byte. The message start identifier is a start byte with the value 0x4D (or `'M'`). The payload content and length is determined by the message type byte, and as error check we use a checksum of one byte. This results in three bytes overhead.

Only important messages, which are not sent periodically, have to be acknowledged. This minimises the communication overhead, and for periodic messages the last message is the most important. So sending the next message is faster than resending the previous one due to an absent acknowledgement. An acknowledgement is only sent by the QR to the PC, and to do so, a specific message type is defined. There are ten message types, for each is given the name, total payload size, whether it needs an acknowledgement and a description.

1. Control message (5 byte): contains a new value for yaw, pitch, roll and lift.
2. Mode change (1 byte, ack): contains mode to which the QR should change.
3. Algorithm parameters (10 byte, ack): contains the 5 controller algorithm parameters: P (yaw), P1 and P2 (pitch/roll), C1 and C2 (pitch/roll kalman filter).
4. Debug message (30 byte): contains a debug string to be displayed on the PC.
5. Telemetry data (26 byte): contains the realtime data to be displayed on the PC.
6. Acknowledgement (1 byte): contains the message type which receiving has to be acknowledged to the PC.
7. Log start (0 byte, ack): requests the QR to start logging.
8. Log request (0 byte, ack): requests the QR to send the recorded log.
9. Sensor log line (28 byte): contains one log entry of the sensor log.
10. Telemetry log line (14 byte): contains one log entry of the status/telemetry log.

4 Implementation

4.1 PC program

The PC program acts as the pilot's control interface: its main feature are reading commands from the keyboard and the joystick, sending them to the QR and visualising the information received periodically by the QR (telemetry data). The core of the PC part of the system is the main event loop, that performs the following actions:

- It checks for an incoming byte from the QR and calls the message handler if it is a message start byte
- If a key was pressed it calls the keyboard handler
- It checks whether the connection timed out (2 seconds) by measuring the time elapsed from the last received message
- It checks whether a certain time has passed since the commands were sent, if so it reads the joystick position and sends it to the QR
- It checks if there are unacknowledged messages in the queue and the acknowledgement timeout has elapsed (100 ms) then it re-sends these messages
- It refreshes the PC program status and the QR telemetry information on the screen

4.1.1 Status

The `status` module contains all the relevant status variables of the system (declared as `extern` variables) that are accessible to the rest of the program. The control parameters variables are also saved in a file when the program terminates. Critical status variables such as commands and control parameters are updated only through specific methods that check for the boundaries of the values, in order to avoid unexpected behaviour.

4.1.2 Communication

The communication-related modules (`sender`, `communication`, `protocol`, `message_handler`) contain the functions to create, send and receive messages accordingly to the protocol described in Section 3.2. If a message needs to be acknowledged (i.e. it is not periodic) then it is also saved in a data structure that can contain only one message per type (the most recently sent). This structure is implemented in the `ack_queue`

module, that provides also the functions for acknowledgement handling and queue resending.
The message handler is called when a start byte is received, and it reads the further message bytes in a blocking way (with a maximum timeout). The telemetry data received by the QR is then used to update the related status variables. The integer values in the payloads require endianness handling because of the difference in the format used on the two architectures (intel architecture on PC and X32 architecture on the QR). In the `communication` module every integer is converted from little-endian to big-endian format during message sending, and vice versa when message reception is handled.

4.1.3 Joystick and keyboard

The `joystick_handler` reads the joystick position, scales the values in a linear way (8-bits integer for yaw, pitch, roll and 10-bits unsigned for lift) and updates the related status variables. The `keyboard` module handles the key presses, used for commands trimming and for sending all the non-periodic messages.

4.1.4 User interface

We realised the `gui` module using the ncurses library[1]. The right windows shows the status, the left windows shows real-time logging information with a color code: green, yellow and red for PC logs, warnings and errors respectively and cyan for QR debug messages.

4.1.5 Program start and completion

Before starting the event loop the program checks if the joystick is in neutral position, uploads and runs the QR executable (through a script execution) and loads the saved control parameters. When the program is closed then it also terminates the program execution on the QR by sending an exit message (implemented as a mode change) and it saves the control parameters. All these functions are implemented int the `boot` module.

4.1.6 Logging

The `logger` module saves the log lines received by the QR in csv formatted files after a log download request is sent. It also logs the messages that are shown on the console during the program execution and the raw bytes received by the QR as hex values. The latter is convenient for debugging communication problems.

[1]https://www.gnu.org/software/ncurses/

4.2 Fixed point arithmetic

To be able to do float calculations on a processor which does not support floating point calculations (on the QR), a fixed point library is required. We have implemented our own library, this to have a library which perfectly suits our needs. The fixed point library has implementations for addition, subtraction, multiplication, division, raise to a power and conversion between multiple precisions. Most previously listed mathematical operations have multiple implementations. One using a function which accepts variables with type `FixedPoint`, and is flexible but slower. The functions check if the precision of both provided parameters are the same, and when different, converts one of them. Further there are also macro versions of the functions, which assume that both parameters have the same precision, but are a lot faster (and therefore end with `_f*`). There are also macro implementations that act directly on integers (in this case the macro name ends with `_fi*`), accept a non fixed-point constant as second parameter (suffixed with `_fc*`) and more precise versions (`_f*p`). The more precise macros take e.g. into account that one of the parameters is very small and therefore shift the value back after mathematical operation instead of before.

4.3 Scheduler

The scheduler is the core of the QR program and links all other QR modules together. It's designed with one specific goal in mind. Ensure that the system executes tasks at a constant phase. It shouldn't matter if the scheduler is not executing code for some time, as long as it ensures, that the filters and control loops are run at a constant speed, and are called as fast as possible when new data is available. This leads to a very simple but stable scheduler shown in Figure 1.

The scheduler has 3 phases witch keep on repeating during the execution of the program. The first state is the "Wait for kick" state. This means that the scheduler waits until the timer interrupt, running at a set frequency, has received the data needed for the execution of the tasks.

At that moment the timer interrupt kicks the scheduler and it will move to the next phase. The next phase is the actual execution phase. This is where all tasks from a task list are executed.

The final phase is the finishing phase. In this phase the scheduler determines which task set has to run next and it analyses the execution time of the ran tasks. If the scheduler detects that the program lags behind more than one iteration, the next loop will be skipped.

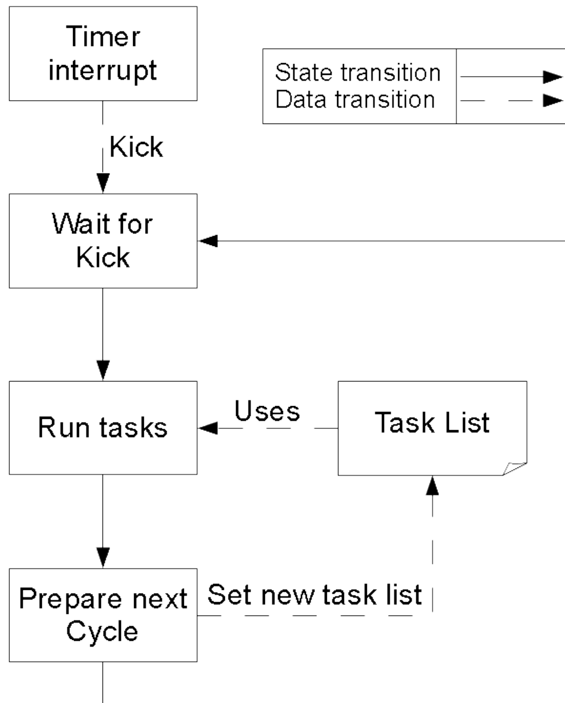

Figure 1: Overview of the scheduler

A different set of tasks is used to satisfy the different modes available on the QR. A few examples of the different task sets are:

- Safe mode: filter, protocol, logger, finished
- Calibration: filter, calibration, finished
- panic mode: Panic, protocol, finished
- Full control: engine control, filter, protocol, logger, finished

4.4 Communication and protocol

All code running on the QR has to be as optimal as possible, therefore no `structs` where used, because they made things around four times slower.

The interrupt which is fired upon receiving of a byte (`uart.c`: `isr_rs232_rx()`) puts the received byte in a circular buffer (which has the wrong name `fifo`). The function `protocol.c`: `_protocol_check_message()` is called from the scheduler which runs in the main loop, and checks if:

- The first byte in the circular buffer is the start byte (0x4D), otherwise this byte is discarded
- If the message type is defined, i.e. the value of this byte is one of the message types
- If the amount of bytes in the buffer is large enough to contain a whole message for the given message type

If so, the function `protocol_parse_message()` parses the message. First the checksum is calculated on the bytes in the receive circular buffer, and if this is right, a setter function of the corresponding module is called, with the payload data as parameter(s).

There are a few options to send a new message to the PC using the protocol, of which only one is used. The other ones were only implemented for testing purposes or use `structs` which is slower. The mainly used method is `protocol.c`: `protocol_send_new_message()`. This function has three parameters: the message type, the length of the message and a pointer to the payload. It directly puts the start byte, message type byte and payload in the send buffer. If the payload length exceeds the defined message payload size, the payload is trimmed. If the set payload length is shorter than the defined message payload size, additional zeros will be send to fill the message. While the function puts all the bytes in the send buffer it calculates the checksum over all sent bytes except the start byte. At last, the checksum is also added to the send buffer. The other functions which can be used to send a message from `protocol.c` are:

- `protocol_create_message()` to create a new message in a `struct` of type `Message`.
- `protocol_send_message()` to send a message created with the previous listed method.
- `protocol_send_fixed_message()` to send a fixed message from the given parameters

To be able to control the amount of processor time it takes to actually send the bytes over the UART (universal asynchronous receiver/transmitter), only bursts of 10 bytes (the value defined in `SND_SEQUENTLY_BYTES`) are sent. The

start of the burst is triggered by the scheduler which runs in the main loop by calling `uart.c`: `uart_start_snd()`. `uart.c`: `uart_putchar()` actually copies the bytes to the UART register, and keeps track of the current bursts byte count in `snd_byte_cnt`.

There are two debug modes in which the protocol and uart modules can work: `DEBUG_MODE_PROTOCOL` and `DEBUG_MODE_DIRECT`. The first uses the protocol while the latter directly prints the debug strings to the UART, and outputs any other message as hex values. This allows one to see whether and where communication goes wrong, and we can see the output in a regular terminal program like minicom when the PC program can not interpret the data.

4.5 Filter

The main goal of the filters running on the QR is to remove as much noise from the sensor values as possible. This has to be done while taking several constraints into consideration:

- Real world time constraint
- Delay of data caused by the filter
- Limited 32 bit architecture

4.5.1 Filter design

The first part of designing a filter is deciding upon what kind of filter to use. We followed the suggestion given during the classes an went for an IIR Butterworth filter. The decision of what order to be implemented came from time constraints. A second order Butterworth filter takes roughly 200μs to execute on our system. This would have lead to the total execution time of 1200μs for all 6 filters. This would later turn out to be too much execution time. A First order Butterworth filter only takes 120μs (720μs total) to execute and was therefore chosen as the final order for the filter.

In order to decide the cutoff frequency several calculations where done in Matlab. The first step was identifying what kind of data we actually needed to filter. For this a log was made while the QR was held in hand and engines rotating at approximately floating speed in order to replicate the noise of an actual flight as closely as possible. An FFT of this log is shown in Figure 2. It shows that there are a lot of peaks in the data at several frequency ranges:

- 0-10Hz: Main physical movements of the QR
- 65Hz: Rotations of the engines
- 130Hz, 195Hz: Higher harmonic frequencies of the engines

Of all this data only one the 0-10Hz section is interesting. The first bit peak of noise lies around 65Hz. This tells us that the filters cutoff frequency should be between 10Hz and 60Hz. A lower cutoff frequency leads to more noise reduction which is good. A higher cutoff frequency leads to less phase shift in the lower frequencies and is also good. For this reason a good trade off needed to be found. Because we run the whole system at 500Hz (the minimum frequency) we value the phase-shift data more than the reduction of noise. For this reason we Tested 11 different cutoff frequencies in matlab. These are shown in figure 2 on the right. The 25Hz filter (light blue) had a good reduction of noise and less phase shifting than its lower frequency counterparts and was therefore chosen as the final cutoff frequency for the filter. This leads to the following filter coefficients:

- a0 = 0.1367 = 0x00000230 (12 bit precision)
- a1 = 0.1367 = 0x00000230 (12 bit precision)
- b1 = -0.7266 = 0xfffff460 (12 bit precision)

4.5.2 Fixed point implementation

The designed filter was implemented on the QR with twelve bits of precision. This is an amount that is not optimised for our filter at and could have been implemented in a better way. The filter also uses some unnecessary operations. All of these optimisation where not made due to time constraints.

The filter could have been implemented with 8 bit precision instead of the 12 bit used. The coefficients do not use the last 4 bits (they are 0) and are therefore useless. The code currently also runs with a lot of unnecessary fixed point math. before the sensor value is used for calculation, it's converted to a fixed point number. then, after the multiplication has been completed this byte shift needs to be reverted with another fixed point conversion. All of the code code could have been reduced the following code block, but this was not done due to implementation problems and lack of time:

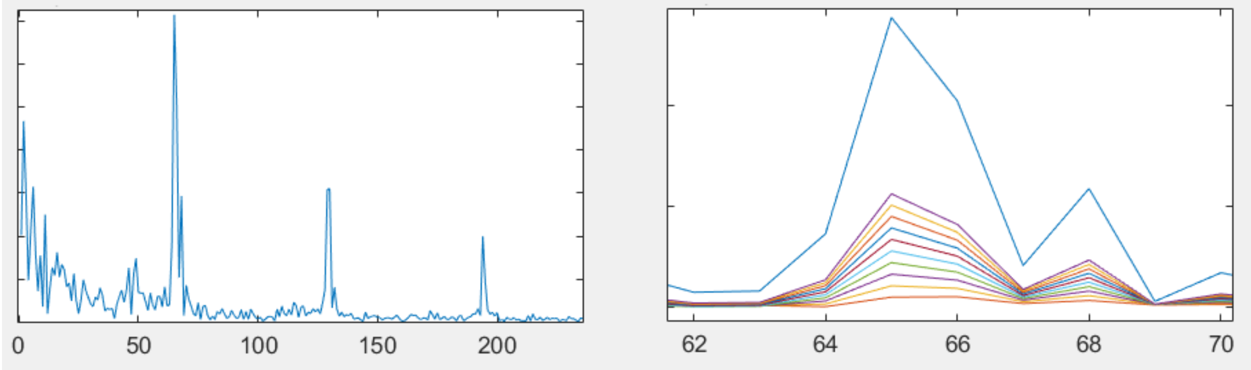

Figure 2: FFT of the data log before and after filtering with different filters

```
1       x0 = sensors - sensor_offset;
2       filterOutput = x0 * A0 + x1 * A1
3                      -y1 * B1;
4       x1 = x0;
```

4.6 Calibration

The `calibration` module is called through the `RunCalibration` method takes the average of the last 100 filters samples for every sensor, and stores them in the `sensor_offsets` array. This array can then be accessed when current calibrated filter values are required, e.g. from the `filter` module. The `is_calibrated()` check is used by the `statemachine` to check if the calibration was performed.

4.7 Controller

On the QR three separate controllers are implemented. One for yaw, pitch and roll. The yaw controller is a rate controller and the identical pitch and roll controllers are attitude or angle controllers.

The controllers are called from the scheduler which runs in the main loop. Only when the QR runs in yaw control mode the corresponding yaw controller is called, and whilst running full control mode all controllers are involved. The controllers use the latest output data of the filters or sensors, and the controller algorithm parameters which are set from the PC program. Using a constant (`DATA_SRC`) it is possible to choose between using direct sensor output (`SENSOR_DATA`) and using filtered data (`FILTER_DATA`).

4.7.1 Yaw controller

The yaw controller is a simple P-controller which controls the rate. The main formula for the controller is $yaw_c = P \cdot (yaw_s - S_{gyro})$, where yaw_c is the corrected yaw value, yaw_s is the yaw set-point set using the joystick on the PC and S_{gyro} the (filtered) output of the corresponding gyro sensor. The controller is implemented using fixed point arithmetic and uses the same precision as the filter to be able to use the filter output directly without converting it to another precision. The value is converted back to a normal integer before passed to the engine mapping module. The engine mapper does only use addition and subtraction and has therefore no need for fixed point numbers.

4.7.2 Pitch and roll controller

The pitch and roll controllers are identical. In the source code they do however not share the same code, this is to make the execution faster. The controller needs three persistent variables and of course there are methods to implement this with e.g. structs, so that there is only one function with the controller code. The code which selects the right variables will slow down the execution of the controller so we have chosen to use two separate functions with their own static variables and with the same code.

Listing 1 shows the implementation in pseudo code of the pitch controller. All variables are fixed point numbers, but to keep this listing simple, all fixed point specific functions are removed. First the gyroscope sensor value is multiplied with a scalar to normalise the value, than the bias is subtracted. Line `3` and `4` calculate the new value for phi, and line `5` the new bias value. The last line calculates the corrected pitch value which is passed to the engine controller, where `pitch` is the set-point set by the PC program.

```
1  rate = FILTER_VALUE(SENS_GYRO_PITCH)
2        * PID_GYRO_SCALAR - p_bias;
3  phi = phi + rate * P2PHI;
4  phi = phi - ((phi - acc_pitch) / C1);
5  p_bias = p_bias +
6        ((phi - acc_pitch) / C2);
7  new_pitch = P1 * (pitch - phi) -
8        P2 * pitch_r;
```

Listing 1: Pitch controller pseudo-code

4.8 Code metrics

The total compiled size of our QR program is 123kB with test mode disabled. With test mode enabled, another 13kB is used for the test data. The C code itself is 97.5kB. The number of lines of code (.c and .h files) is 3587. The size of the PC program executable is 61,5kB and the PC C code is 49,1 kB. The amount of lines of code is 2044.

5 Experimental results

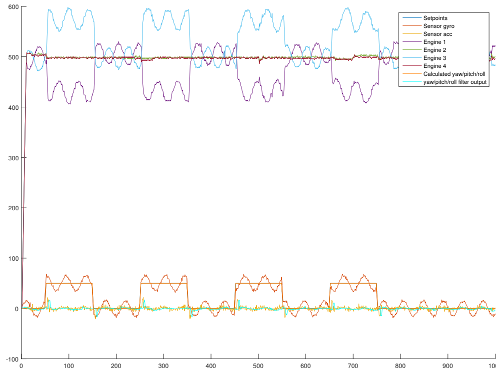

Figure 3: Output of a test run

5.1 Loop frequency

In order to measure the performance of the system, several timers where added to the code of the scheduler module. They measure the time the code spends in each component. A overview of these times can be seen in table 2. With these times it was possible to estimate how fast several schedules could run. The longest schedule available (Full Control with logging) takes approximately 1400μs. The program was set to run at 500Hz (2000μs per iteration). This means that the program is able to run the full control schedule without any problems or missing of deadlines. This was also seen as the "deadlines missed counter" remained at a steady number after start up.

These results suggest that it's possible to raise the frequency of the scheduler. Especially as the QR is normally not logging any data (reducing the average loop time to approximately 1200μs). It was attempted to raise the loop frequency to 750Hz (1000 000 / 750 = 1333.3μs per loop) as this should be feasible. Experimental evaluation showed different results. The average loop time was as expected around 1200μs and yet was the "deadlines missed counter" increasing rapidly. This was caused by the communication between the QR and the PC. When the processor receives or sends a message, it results in a big spike in the processor load. This spike causes the program to miss several deadlines in a row.

It might be possible to increase the loop frequency if more time was available for the project team. Then several tasks could have been optimised in order to create a faster system. For the final implementation the ideal loop frequency remained 500Hz, as at this values, there where very little deadline misses.

Task	Average execution time
Filters	674μs
Full Control	455μs
Yaw Control	265μs
Manual Control	208μs
Logger	210μs
Protocol	80μs

Table 2: Average latency of the program while executing several code blocks

5.2 Test mode

To be able to test the controllers and the filter at home on the development board, we implemented a test mode. When test mode is enabled (by defining the constant `TEST_MODE`) and test data is set in `pidtester.c` it is possible to automatically run a test. Hereby the set point data (`test_setpoint_data`) is used to set a certain set point for yaw, pitch or roll. Also the sensor data is set to the global `sensors` array using the filter test data (`test_filter_data`) from `pidtester.c`. The `sensors array` is normally set in the `engines.c` file, and uses the interface `pidtest_get_sensor()` to obtain the test data. See Figure 3 (page 7) for the output of a test

Test performed	Result
The QR has a safe state where the engines won't turn	Passed
The safe state can be reached at all times	Passed
The QR cant leave the safe state when controls are not zero	Passed
The program wont start if the Joystick is not in neutral position	Passed
The system is able to make a emergency stop at all times	Passed
The system can reach the emergency stop via the '1' key	Passed
The system can reach the emergency stop via the joystick trigger	Not passed
The engines are not allowed to stall during operation	Passed
Sensor data must be filter with a digital filter	Passed
The communication between the QR and PC must be reliable	Passed
Losing communication results in an emergency stop	Passed

Table 1: Safety checks performed during the final demonstration.

run. The top lines are the engine values, the block wave are the pitch set-points. The acceleration and gyroscope sensors are the other two lines which have, on purpose, some sinusoidal noise.

5.3 Safety Tests

In order to see if the system was safe enough to take flight with as little risks as possible a lot of tests where fulfilled. These tests shown that our system fulfilled the safety requirements can be seen in table 1. A quick look shows that only one requirement was not fulfilled. This was caused by a misinterpretation of the requirements.

5.4 Demonstrator capabilities

Al safety requirements are implemented as listed in Table 1. Further manual mode works fine, the logger works for both sensor data and status/telemetry data. The communication link is stable, which means that only very occasionally we get a checksum error and a discarded message. The scheduler which schedules all tasks at 500Hz is also working good, which means that we do not have a lot of deadline misses. The filter is also working fine.

Only the controllers did not work as expected. As pointed out by the teaching assistants during the final test, was the feedback of the controller working in the opposite direction. This means that every movement is strengthened instead of counteracted. When we look more closely to the result of the test mode output, we can indeed see that the controller makes the engine values even higher if the sensor value raises above the set-point. This should not be the case.

6 Conclusion

The final design is divided in clear modules. Only while developing we did not create each module on it's own and test is separately from the other code. Most modules where tested in combination with each other. This results in a system where the interfaces of the modules are less defined. On the other hand do all the modules have a clearly defined task. The protocol is lightweight as it should. The system base can be said to be done and super stable.

The only part of the project that is not done yet is full control, but we we are almost there. The result as a team is therefore pretty good. Everybody has invested around the same amount of time in the project. Initial c programming skills differed a bit and so might the resulting amount of code.

Such a project is a good way to learn developing software for embedded systems with low resources. Also collaborating on a shared code base gives insight in e.g. integration problems. The hardest part of the project was the limited availability of the QRs itself. No time could be wasted on testing and debugging code during the time the QR was available to the team. This has lead to the invention of many creative ways of testing code without the hardware itself. In hindsight we can say that this was a fun project to work on even though it was super tough and a lot of work.

Appendix A - Software architecture and modules

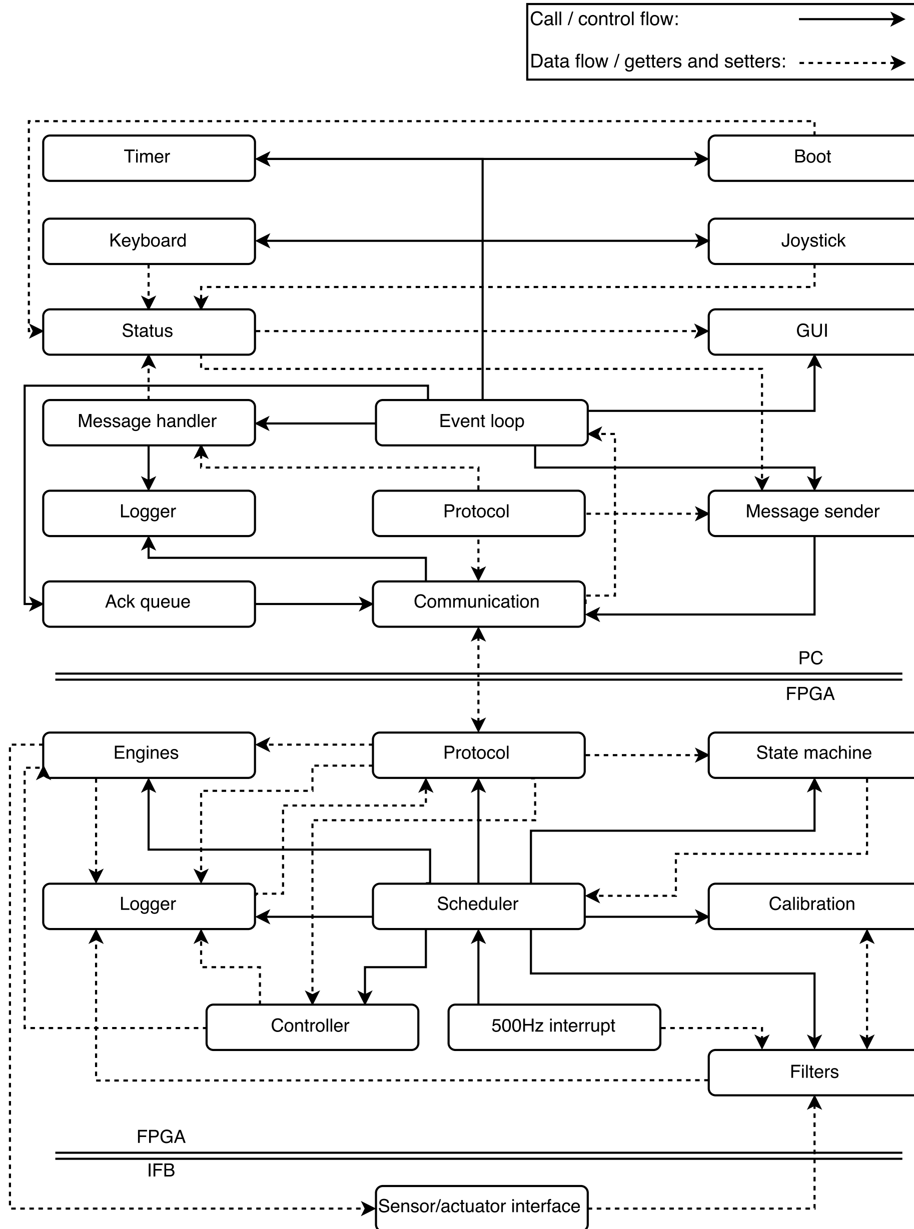

Figure 4: Software architecture

PC program	
Module	**Interfaces**
ack_queue	void insert_into_ack_queue(Message *m); void insert_into_ack_queue(Message *m); Message *acknowledge(); void resend_ack_queue(); int is_ack_queue_empty(); void print_ack_queue();
boot	void init_program(); void end_program(bool conn_lost);
communication	void term_initio(); void term_exitio(); void open_fpga(); void close_fpga(); int fpga_getchar_nb(unsigned char *c); int fpga_getchar(); void send_message(Message *m, int ack_enqueue); Message *receive_message(); void start_fpga_program();
gui	void gui_init(); void gui_exit(); int kbhit(); void print_status(); void print_scroll(char *fmt, int color_pair_id, ...);
joystick_handler	void joystick_init(); void handle_joystick(); void close_joystick();
keyboard	void handle_keyboard_input(int ch);

logger	void init_logger(); void log_sensors_line(PayloadLogSensors *p); void log_telemetry_line(PayloadLogTelemetry *p); void log_console_line(char *str); void log_incoming_byte(unsigned char c); void loghex_newl(); void exit_logger();
message_handler	void handle_message(Message *m);
protocol	void dump_msg(Message *m); Message *new_message(uint8_t type, void *payload); unsigned char calculate_checksum(Message *m); PayloadControl *new_payload_control(uint16_t lift, int8_t roll, int8_t pitch, int8_t yaw); PayloadMode *new_payload_mode(uint8_t mode); PayloadParam *new_payload_param(uint16_t p, uint16_t p1, uint16_t p2, uint16_t c1, uint16_t c2);
sender	void send_params(); void change_mode(uint8_t md); void send_commands(); void start_logging(); void log_request();
status	ad_status(); void save_status(); int8_t trim_up(int8_t trim); int8_t trim_down(int8_t trim); void lift_trim_up(); void lift_trim_down(); uint16_t param_up(uint16_t par); uint16_t param_down(uint16_t par);
timer	void start_timer(struct timeval *t); double timer_value_ms(struct timeval *t); double timer_value_us(struct timeval *t); void update_response_time(struct timeval *s);

Table 3: PC program: modules and interfaces.

QR program	
Module	**Interfaces**
EngineControl	short getCurrRoll(void); short getCurrPitch(void); short getCurrYaw(void);
LazyTimer	void initTimer(LazyTimer * timer); void ResetTimer(LazyTimer * timer); int TimePassed(LazyTimer * timer); int runTimer(LazyTimer * timer, int waitTime); int TimePassed_us(LazyTimer * timer); int runTimer_us(LazyTimer * timer, int waitTime);
SensorLogger	void RunSensorLogger(void); bool SensorLoggerRun(void); LoggerStatus SensorLoggerStatus(void); void SensorLoggerStart(void); LoggerStatus SensorLoggerSendLog(void); void printlogsendstatus(void);
status_logger	nputs(uint16_t l, int8_t r, int8_t p, int8_t y); LoggerStatus status_logger_state(); void status_logger_start(); void log_status_line(); void print_status_log_send_progress(); LoggerStatus status_logger_send_log()
calibration	void RunCalibration(void); void print_offsets(void); bool is_calibrated(void);
debug	void debug_printf(const char*, ...);
engines	bool RunPanic(void); void _updateEngineValues(void); void Engines_init(void); void SetEngineValues (unsigned short e1, unsigned short e2, unsigned short e3, unsigned short e4); void EnginePANIC(void); bool EnginesOff(void); void printEngineValues(void); unsigned short getEngine1(void); unsigned short getEngine2(void); unsigned short getEngine3(void); unsigned short getEngine4(void);
exceptions	void exceptions_test(void); void exceptions_init(void);
filter	void filter_init(void); bool runButterworth_2_10(int SensorNr); void RunFilters(void); void runFiltersFull(void); void skipSampleFilter(void); int getFilterLack(void);
fixedpoint	void fp_conv(FixedPoint* A, int precision); void fp_print(FixedPoint *A, char *str); /*Fixed point arithmetic macros*/
Scheduler	void setSchedule(Tasks * newSchedule); void resetTimingStatistics(void); void startProgram(void);
pidcontroller	void pid_set_params(int, int, int, int, int); void pid_yaw_control(void); void pid_full_control(void);
pidtester	int pidtest_get_sensor(int); void pidtest_set_next_setpoint(); void pidtest_send_log();
protocol	void RunProtocol(void); bool protocol_check_message(void); bool protocol_parse_message(void); void protocol_send_fixed_message(byte, int, ...); Message* protocol_create_message(byte, int, void*); void protocol_send_message(Message*, int); void protocol_send_new_message(byte, int, void*); void protocol_ack(char); bool protocol_buffer_empty(void); int protocol_message_size(int);
statemachine	ProgramState getMainState(void); void PANIC(void); void RunStateMachine(void); void setState(uint8_t);
uart	void uart_init(void); int uart_outbuffer_space(void); int uart_buffer_size(void); int uart_outbuffer_size(void); bool uart_delete_byte(int); void uart_print(void); void uart_print_outbuffer(void); int uart_checksum(int, int); void uart_send_byte(byte); void uart_putchar(void); void uart_start_snd(void);

Table 4: QR program: modules and interfaces.