Technical Report

Team H

Victor Spiridon	Vlad Dumitrescu	Răzvan Venter	Matei Ţene
Computer Science	Computer Eng.	Electrical Eng.	Applied Mathematics
4184599	4185846	4184823	4182022

{v.spiridon, v.dumitrescu, r.venter, m.tene}@student.tudelft.nl

April 2012

Abstract

It takes a very experienced user with quick reactions to *manually* pilot a quad-rotor helicopter (QR). To enable usage by casual users, we implemented roll, pitch and yaw feedback controllers for automatically stabilizing the QR. This is backed up by a PC user interface through which the user can tune and control the QR through a dependable communication protocol. We especially aimed for low latency between the user's input and QR's reaction, stability but also for reliability and code modularity. Besides keyboard and Joystick, a special user input module was developed for Android phones through Bluetooth.

As notable results we achieved a fast control loop ($< 1ms$), a safe landing procedure in panic mode, very fast QR reaction time, and fast parameter tuning (without recompilation) through a special user interface. The conclusions of our efforts are a very stable QR with just a small drift caused by the gyroscopes and last but not least a great team learning experience.

1 Introduction

In this project we tackle the design and implementation of a software suite for controlling a quad-rotor helicopter (QR) through a PC. It is well known that only experienced pilots can react fast enough to counter angle perturbations that would, otherwise, lead to a crash. To make the flight stable, feedback controllers are implemented for roll, pitch and yaw using the 3-axis accelerometers and gyroscopes. Sensors are not perfect: accelerometer input suffers from noise (mainly caused by the running engines) and gyros have a small resolution (4 bits) and drift over time. To obtain usable sensor values, Butterworth and Kalman filters are used. Because the QR has a very limited processing power, special care has to be taken in dividing its workload such that the control loop is executed at least every 2 ms. Because the X32 soft core installed on the QR's FPGA does not have floating point support, real values have to be represented in fixed point precision using 32 bit integers. To ensure reliability, values are scaled to avoid overflow with possibly disastrous consequences. Also dependability must be ensured e.g., engine updates that prevent engine stalling, faulty PC-link detection, dependable communication protocol. In case of an unrecoverable error, the QR must go to *panic mode* in which it will automatically land on the ground.

The user can control the QR through the keyboard, a joystick and an Android mobile phone connected to a PC. The PC aggregates and sends these user inputs (setpoints) along with other parameters to the QR through a communication protocol via an unreliable serial link. The protocol and the software on the QR is optimized such that the delay between the user action and the QR's reaction is minimal. The software suite is tested offline using a Nexys board in order to adhere to the principle *first-time-right* as much as possible. In the following sections we are going to the discuss the software architecture, implementation details of the various modules, the results, conclusions and future work prospects.

2 Architecture

Fig. 1 depicts the general overview of the system. Going from left to right, the keyboard, joystick and phone input is collected by the Control UI, which forwards commands to another process handling protocol packing. The Protocol packs/unpacks the commands into the appropiate format and sends these to the serial/wireless link. The Control Protocol Task reads the packets and turns them back into commands for the QR. The Telemetry Task sends data back to the PC, while the Control Loop Task constantly runs the needed controllers based on the current QR's state. The 3 tasks on the QR are scheduled in a round-robin fashion with 4 slots. The Control Loop runs in slots 1 and 3, the Telemetry runs in slot 2 and the Control Protocol runs in slot 4

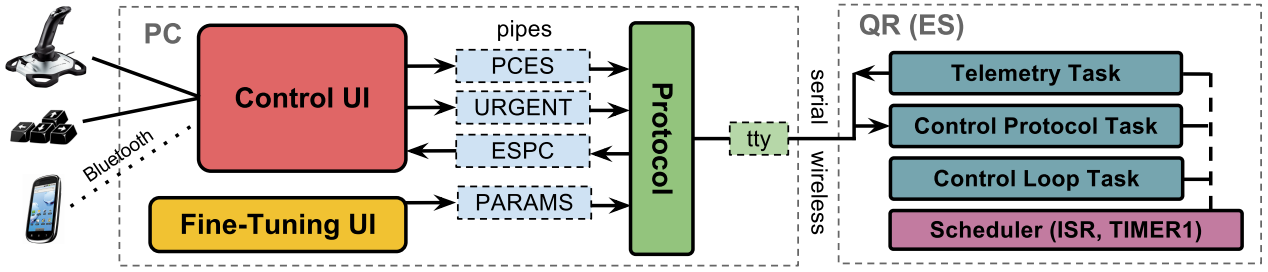

Figure 1: Architectural overview of the main software components (UIs, Protocol and QR).

(i.e., control loop → telemetry → control loop → control protocol → control loop and so on). Different timings for each slot are possible.

The UI processes and the Protocol process are interfaced using 4 pipes. The PCES pipe is used for data going from the Control UI to the QR, the URGENT pipe is used for important data for which any unnecessary processing delay should be avoided (e.g., the PANIC mode command), the ESPC pipe will forward data comming back from the QR (e.g., telemetry), while the PARAMS pipe is used during development and tuning to send different parameters (e.g., Kalman constants, Butterworth coefficients) from the Fine-Tuning UI (a separate application).

Sensor filtering is done in two stages: low-pass filters are executed at data acqusition (1300Hz), while Kalman filtering is done at the beginning of the control loop (1000Hz).

3 Implementation

3.1 The User Interface

The User Interface (UI) is implemented in C in text mode. The refresh rate of the interface is 100Hz, therefore $\mathbf{r}$ = 10ms is the smallest delay in which new information can be read from the user input. The information printed on the screen is refreshed every $\mathbf{p}$ = 50ms (20Hz). The rate was chosen as the best compromise between user response time and processing overhead. At startup, the interface resizes the terminal window in which it runs to 30 rows and 80 columns if it is smaller than these values. Note that the resize works only with `gnome-terminal`. Every $\mathbf{r}$ milliseconds the UI processes the user input and takes the decision whether to send a new packet to the protocol or not. Additionally, if a new packet has not been sent in the last $\mathbf{k}$ = 100ms the interface will re-send the current mode, as it is the smallest packet, in order to keep the connection alive. In the wireless mode the UI re-sends the setpoints instead of the mode, to counteract the higher probability of undelivered critical setpoint packets. The packets are sent to the protocol through the `PCES` pipe.

3.1.1 Mode Change

The interface starts in *safe mode*. In order to exit *safe mode* and enter *calibration*, *yaw control* or *full control mode*, the setpoints (lift, roll, pitch, yaw) have to be on **0**. The *panic mode* can be entered from any mode. The *panic mode* packets are sent through the `URGENT` pipe to the protocol, to avoid any congestion on the `PCES` pipe. The UI stays in *panic mode* for 1 second, then resets the setpoint offsets to **0** and enters *safe mode*. After initiating the *panic mode*, the user should also wait for the QR to enter *safe mode*. Because the mode is sent constantly, the QR will always stay in *calibration mode* if not specified otherwise. Therefore, after the desired calibration duration (at least 3 seconds) the user should manually exit *calibration mode*.

3.1.2 User Input

Through the keyboard the user can set the offsets for the lift, roll, pitch, yaw, the mode, and the feedback controller Ps individually for roll, pitch and yaw. The key input is read in non-blocking mode from `STDIN` and the key mappings respect the assignment's specifications. The roll, pitch and yaw are bounded to the values **[-126, 126]** and the lift and Ps to **[0, 126]**. The Ps can be saved to a text file, in which case they are automatically loaded on startup.

The Joystick is also read in non-blocking mode, and the setpoints are mapped to the bounds stated above. The Joystick setpoints are summed with the offsets. As required, the fire button initiates *panic mode*. The UI displays if the Joystick is connected or not.

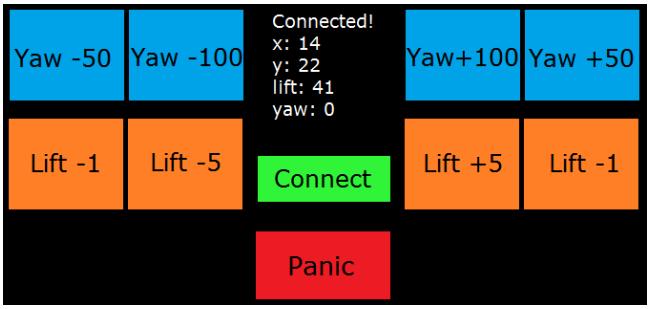

Figure 2: Screenshot of the Android App. OX axis is for pitch, OY axis is for roll.

In addition, an Android app was developed as a new input method that takes advantage of the sensors and touch screen of a smartphone (ZTE Blade). Communication is done through Bluetooth (see Fig. 2). The roll and pitch are obtained from the phone's accelerometers. The yaw can take the values **-100, -50, 50, 100** by long pressing the yaw buttons, otherwise is set to **0**. The lift can be adjusted via buttons that deliver the increments: **-5, -1, 1, 5**. The lift buttons can be clicked, or long pressed in which case, the increments are applied at 200ms delay. The values sent are bound to the UI intervals stated above and the app also supports entering *panic mode*. Before the connection, the phone and the PC have to be paired (successfully tested on Ubuntu 9 and 10). At the PC side a RFCOMM socket is created that can only be read in blocking mode. Therefore, a dedicated thread has been assigned to read from the socket and also measure the delay between packets. If the delay exceeds 1 second, the connection is considered dead and the UI displays the Bluetooth status as disconnected. Otherwise the UI displays the rate at which it receives data from the phone.

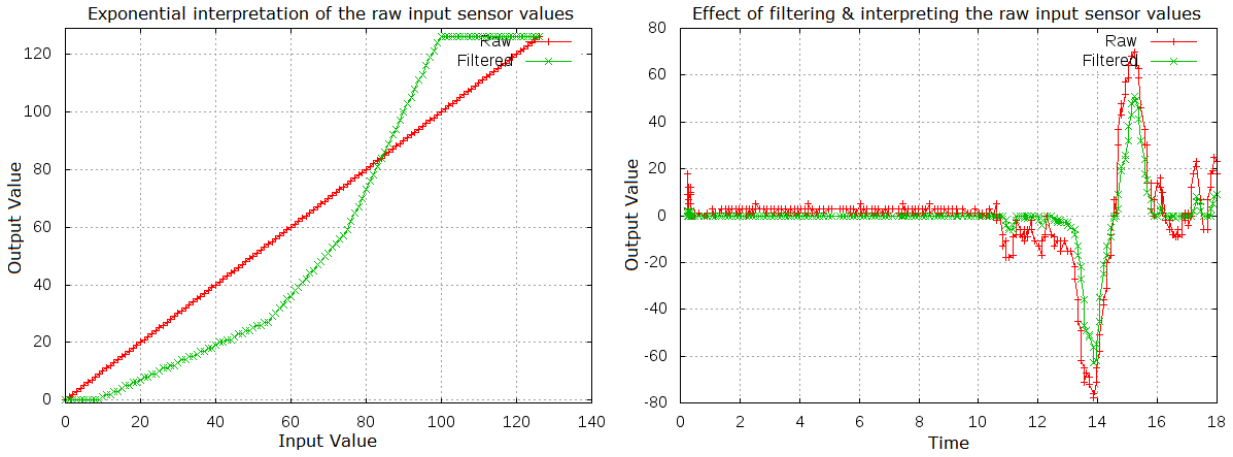

Figure 3: Left: Exponential interpretation of the sensor values from 0 to 126. From -126 to 0 the graph is mirrored. Right: Filtering and interpretation results on real data: 320 samples over 18 seconds.

The highest sampling rate that has been achieved was around 17 samples/second (probably due to the low quality of the phone). Usually, the samples come in pairs of two at 1 to 5 ms interval and the interval between the pairs varies between 50 and 200 ms. Moreover, the sensor values have a small noise (1 or 2 units) at a frequency close to the sampling rate (rarely two consecutive samples are identical). To overcome this, at the PC side an average over the last 3 values is made. This was found to be the perfect compromise between the responsiveness and noise. Note that the Bluetooth communication also introduces about 300ms of delay. Our implementation has an exponential interpretation of the roll and pitch values (see Fig. 3). As a result, the **0** setpoint can be obtained in a wide range around the horizontal position, to counteract the noise (and possibly a shaky hand), such that the *safe mode* can be easily changed. Tilts near horizontal are not that sensitive such that the user can have a fine control over the QR. At tilts that exceed 45° the values increase more rapidly, and reach the maximum around 80°. A maximum mapped to a tilt of 90° would have been too inconfortable for the user's wrists. Each sample, along with the filtered values, is logged on the PC.

3.1.3 Telemetry Data

The UI reads data sent by the protocol from the QR from the `ESPC` pipe, in a non-blocking fashion. When the QR is connected through the serial cable, telemetry data is sent every 100 ms. This data contains the current mode of the QR, the length of the control loop, the filtered sensor values for the accelerometers and gyros, and the engine values. The UI always displays the last 10 data sets by means of a circular buffer. The data along with the setpoints and the Ps are logged on the PC.

3.1.4 Parameter Fine-Tuning GUI

In order to adjust the various parameters of the control and filter algorithms without recompilation, a Graphical UI in Java was developed. More precisely we can activate individually: the Butterworth filters for the sensors and the P controllers. Also we can adjust the Kalman coefficients C1 and C2 and the Butterworth order, sampling rate and cutoff frequency. The coefficients are computed according to the code at [6]. We used this interface only in development, afterwards a set of standard parameters are loaded from a file on the PC.

3.2 Communication Protocols

We differentiate between the two data streams that need to be handled: the one going from the PC to the QR and the one going from the QR to the PC. The former stream is handled by the **Control Protocol (CP)**, while the latter is handled by the **Telemetry Logging Protocol (TLP)**. Underneath CP there is another layer, named Data Link (DL), that ensures realiable communication over the unreliable serial/wireless link that is used.

As Fig. 4 shows, CP supports 6 types of messages. The first 3, mode change, attitude change, and parameters change are required by the Control UI to change the QR's mode, setpoint (i.e., lift, roll, pitch, yaw), and, respectively, the parameters for the 5 controllers (yaw rate, pitch rate and angle, roll rate and angle). The last 3 messages are used by the Fine-Tuning UI. The **Activate** flag bits are used to enable or disable the Butterworth filters, the roll and pitch controllers, and change the signs of the controllers' feedback. The Fine-Tuning UI can compute new coefficients for the Butterworth filters using different cut-off frequencies. These can be sent to the QR using the **Low-pass Coefficients** message. There are 3 sets of 3 coefficients (e.g., for the yaw gyro, for the other gyros and for the accelerometers filters). The constants for the Kalman filters can be changed using the **Kalman constants** message.

3.2.1 Packet Formats

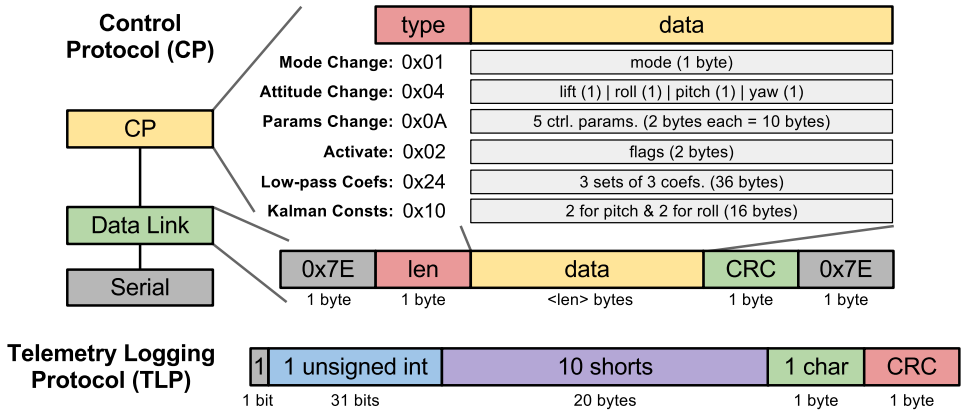

Figure 4: Protocol packet formats.

The **Data Link** layer is similar to HDLC. Each DL frame starts and ends with `01111110`. After this start byte, the length of the data part follows, while the end byte is preceeded by a 8-bit CRC checksum of the `len` and `data` fields (the `0x07` polynomial is used). If the `0x7E` or `0x7D` bytes are encountered inside the frame, they are escaped by prefixing them with `0x7D` (the escape byte) and inverting the 5th bit of the original byte.

Because all message types have different lengths, the **Control Protocol**'s **type** field overlaps the DL's **len** field. While DL uses this byte to check that the received chunk is actually a valid-sized frame, the CP uses it

to distinguish between the message types. For each message type different values are encapsulated in the **data** field (see Fig. 4).

The **Telemetry Logging Protocol** does not use the Data Link layer, but has a XOR-based CRC checksum. During our tests we did not encountered any problems with lost or corrupted bytes from this stream. Also, even if bytes are lost or corrupted, in some rare occasions, it would not critically affect the QR's operation or controlability like in the CP's case.

3.2.2 Technical Details

The PC implementation multiplexes the I/O operations on the 4 pipes and the TTY device using libev [7]. Basically each input channel has a associated callback function that is called when data is available. After receiving data on the PCES, URGENT (for transioning to PANIC mode), or PARAMS pipes (i.e., some serialized C structs), the specific protocol packing functions are called and the resulting packet is written on the TTY. Data received on the TTY (TLP packets) is unpacked and send as a C struct via the ESPC pipe to the Control UI.

On the QR program, the bytes received from the PC are initially stored in a FIFO buffer by the RX interrupt (serial or wireless). Two tasks (see 1) are dedicated to communication. The **Control Protocol Task** reads DL frames, delimited by the start and end bytes, from the FIFO, checks their length and CRC (table-based), unpacks them based on their CP type, and calls a specific function with the new command (e.g., `change_mode()`, `change_attitude()`). These functions are in charge of enforcing specific rules (e.g., state machine transitions). The **Telemetry Task** builds a TLP packet (i.e., with the current timestamp or some other measured time, sensor values, motor values and mode) and writes directly to the peripheral.

3.3 Fixed Point Arithmetic

Integers do not offer enough precision when filtering sensor input. Unfortunately the X32 soft core does not have a floating point unit [2] and using a software floating point library [3] is too CPU intensive for our application. Our solution was to implement a small library for integer fixed-point operations using the Q number format [4].

The Q number format uses the first **n** *least significant bits* of an integer for the fractional value (*n-fixed-point precision*, abbreviated Q_n). More precisely, the stored value is the original one multiplied by 2^n, while any remaining decimals are truncated. Consequently, we can use the native support for negative numbers, addition and subtraction, while multiplication and division have minor overhead, as described in the following.

3.3.1 Multiplication and Division

These two operations have to be performed in *2n-fixed-point precision* (Q_{2n}) to maintain accuracy. Multiplication is first done in the usual manner, which results in a Q_{2n} number ($value \times 2^{2n}$) that needs to be converted back to Q_n by an n right bit shift (i.e. division by 2^n). In the case of division, the first operand is brought to Q_{2n} format using an n bit left shift (i.e. multiplication by 2^n), before perfoming the usual operation. Note that the format conversion is performed by truncating the extra n fraction bits. We have used an intermediary step to perform rounding: adding 2^{n-1} after multiplication and half the second operand before division, respectively (as described in [4]).

3.3.2 Precision

As stated in the manual [2], there is no 64 bit support on the X32 processor. Therefore, we have decided to use the Q_{11} fixed point format: *1* sign bit, *20* bits for the integer part and *11* for the fractional part. This means that values which undergo mulitplication or division must reside in $(-1024, 1024)$.

3.4 Filtering

In order to control the movement of the QR, it is vital to know at each particular moment its position and behavior. For each of the three axes, this can be done by reading the output given by one accelerometer and one gyroscope placed on it. However, as the accelerometer data is very noisy, especially when the motors are running, and the gyroscope has a constant drift, filters have to be designed in order to get the correct information. Two major types of filters were used, a Butterworth low pass filter and a Kalman filter.

3.4.1 Low Pass Filters

Low pass filtering is very useful in reducing the noise that is covering the useful signal. Butterworth filters were selected for this application, as they have very low attenuation in the pass band (close to 0dB). Rejecting the unwanted frequencies can be done by carefully choosing the cut off frequency. Considering the fact that the Butterworth filter rolls off more slowly around the cut off frequency than compared to other linear filters, such as Chebyshev or Elliptic [5], the cut off should be chosen somewhat lower than the first unwanted frequency. In order to design the filter, we first started by analyzing the frequency spectrum of an unfiltered accelerometer. This is presented in image (a) of Fig. 5.

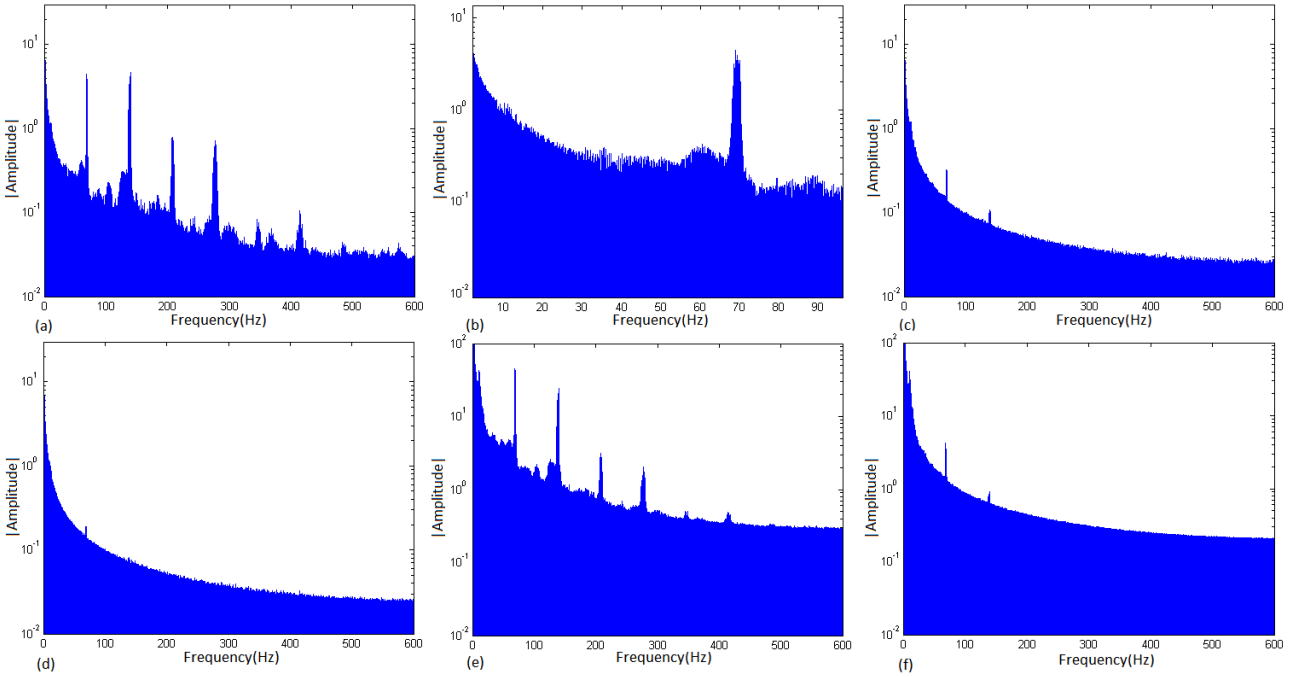

Figure 5: (a): Frequency spectrum of an unfiltered accelerometer. (b): Zoom-in image of the raw accelerometer, representing the useful information frequencies. (c): Frequency spectrum of accelerometer filtered with 18Hz cut-off low pass filter. (d): Frequency spectrum of accelerometer filtered with 10Hz cut-off low pass filter. (e): Frequency spectrum of the Kalman filter output, without filtered accelerometer input. (f): Frequency spectrum of the Kalman filter output, with filtered accelerometer input.

It can be seen that the useful information is found at low frequencies, see image (b) of Fig. 5 , while a lot of noise is added, starting with around 50Hz and moving to higher frequencies. Considering this, we thought about choosing a cut off between 10 and 20Hz. The spectrum of the signal filtered with a 18Hz and then a 10Hz low pass filter is presented in images (c) and (d) of Fig. 5. It can be seen that both of them cancel almost all of the unwanted noise. However, experimental results showed that the 10Hz low pass filter resulted in periodical unexpected behavior of the QR, because some of the useful information was also removed. Therefore, in the end, we decided to keep the cut off at 18Hz, as it removes noise and also keeps all of the necessary data

3.4.2 Kalman Filter

If the accelerometer and the gyroscope would have been able to return perfect indications, then these could have been used directly in the control sequence. However, since the accelerometer has noise and the gyroscope has drift, we tried to use the Kalman filter to combine them and get correct results. Basically, the goal of the Kalman filter is to deliver a signal that has the low noise of the gyroscope, and the position precision of the accelerometer. The filters outputs two values, the first being the angle (in our case ϕ and θ), and the angular rate (p and q). First of all, the two inputs of the filter, the angular speed and the tilt given by accelerometers need to be translated into angles. For the accelerometer this is easier, because of the linear correspondence between the angle and the numeric indication. In order to convert the accelerometer indication into an $\pm$ 90° angle, we did a test movement with the QR, moving it to $\pm$ 90° on each axis. In this way we determined the proportionality parameter, named A2PHI in the code, to have a value of 0.83. In the final code, all the constants and parameters were multiplied with 2^{11}, representing the fixed-point precision.

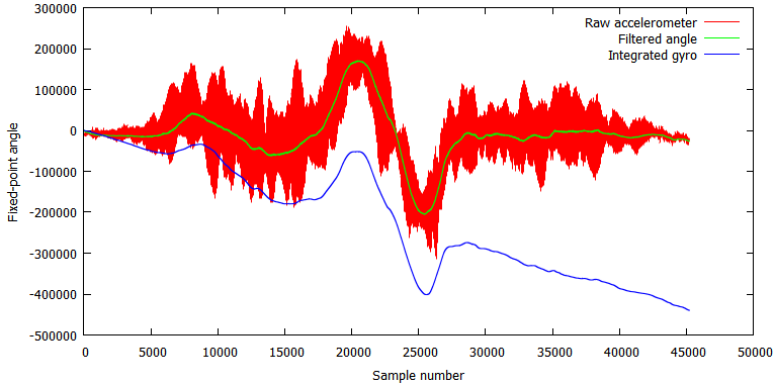

Figure 6: Angle returned by the accelerometer, integrated angle returned by the gyroscope (showing the drift), and angle returned by the Kalman filter (no more drift).

For determining P2PHI, we correlated the indications of the accelerometer and the gyroscope. We made the sum of all the gyroscope rates starting from the beginning (QR on ground, so 0 degrees angle) until the moment when the accelerometer indicated 90°. Then we divided this sum to the total number of samples that were taken and the final result for P2PHI was 0.0029. By plotting the accelerometer and gyroscope returned angle, we tested the correctness of this numbers(see Fig. 6).

However, as it can be seen from Fig. 6, the gyroscope has a strong drift. By choosing the C1 constant of the Kalman filter to the value of 500, the final filtered angle has the precision given by the accelerometer and the low noise of the gyroscope, as it can also be seen in the Fig. 6.

Also, using the second constant, C2, the bias of the gyroscope is adjusted every cycle. However, we decided not to start from an initial bias of 0, but to give an initial offset. This offset was given by the value of the gyroscope during Calibration Mode. In this way, the Kalman filter needs less time to adjust the bias to a useful position, as it can be seen in Fig. 7, which led to better practical results.

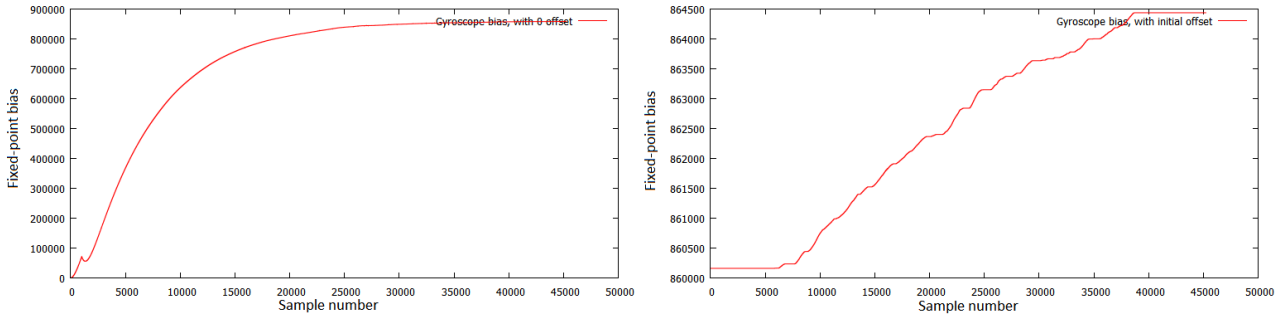

Figure 7: Left: Gyroscope bias variation, with 0 initial offset. Right: Gyroscope bias variation, with a non zero initial offset, determined during Calibration Mode.

3.4.3 Low Pass and Kalman Integration

After designing the Butterworth low pass filter and the Kalman filter, we thought that a good idea might be to filter the accelerometer before applying it at the input of the Kalman filter. Even if the Kalman filter is itself a low pass filter, it still has a small amount of noise in the output signal. By filtering the accelerometer before, this additional noise was completely cancelled, as seen in images (e) and (f) of Fig. 5. Also, no unexpected bad behavior appeared during the simulations, so we decided to keep this structure in the final desing.

3.4.4 Total Number of Filters

In the end, we used a total of three Butterworth low pass filters. Two of them were filtering two accelerometers, before applying them to the Kalman filter, and the third filtered the Z axis of the gyroscope, for Yaw Control. Two Kalman filters were used each control loop, returning the value of ϕ and p, respectively θ and q. For the

Butterworth filters we had the possibility of changing the important parameters (such as the order, constants, cut-off frequency) during the simulation, by using the Fine-Tuning UI. However, in the end we decided that the best results are obtained with second order Butterworth filters, with cut-off frequencies of 18Hz.

3.5 The Controller

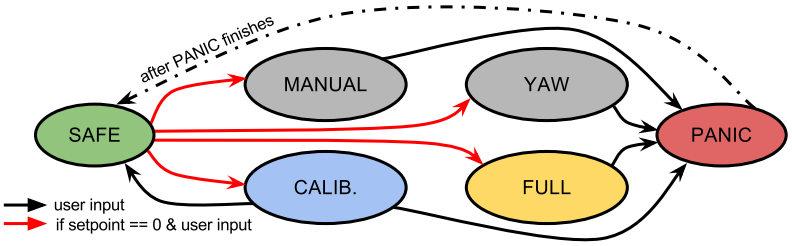

Figure 8: QR operation Finite State Machine

The control loop behaves according to the QR's current mode of operation. The possible modes and transitions between them are depicted in Fig. 8.

After power-up, the QR enters *safe mode*, in which it ignores any received setpoints, keeping all motors at 0 RPM. Leaving this state towards one of the four operating modes (calibration, manual, yaw and full control) is conditioned on having all setpoints on 0.

Panic mode is reachable from all operating modes, either through user input or when an error occurs. In this state, the QR ignores all commands and, once the panic sequence is over, it automatically enters safe mode.

3.5.1 Calibration Mode

During calibration, sensor values are monitored in order to obtain bias and drift rate values. The bias values are obtained by averaging 1024 samples from each sensor, taken at 1ms delay. The gyroscopes are known to have a drift in time, which is assumed to have a linear behavior. To estimate the drift rate, 128 samples from each gyro are collected before and after the bias computation loop. Each set is then averaged and the difference is divided by the elapsed time to obtain the drift rate per 1us for that particular gyro.

3.5.2 Panic Mode

In panic mode the goal is to land the QR safely on the ground. The first step is to stabilize the QR, then bring it to hover position and, finally, land.

Stabilization is done by bringing all motors to the average RPM as quickly as possible (at the same time, avoiding stalling). If the average RPM is higher than the predefined hover RPM, then all motors are synchronously decelerated towards it.

Hover speed is maintained for 0.5s, after which the landing procedure starts. This done by slowly decreasing RPMs towards half the hover level over a period of 4s. After this, the QR is assumed to have landed so engines are stopped.

3.5.3 Motors Update

The behavior of the QR propellers (ae_i = RPM of motor i) in order to obtain the desired effect, according to the four setpoints (lift, roll, pitch, yaw), is governed by the following system of equations:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} {ae_0}^2 \\ {ae_1}^2 \\ {ae_2}^2 \\ {ae_3}^2 \end{bmatrix} = \begin{bmatrix} lift \\ roll \\ pitch \\ yaw \end{bmatrix} \implies \begin{bmatrix} {ae_0}^2 \\ {ae_1}^2 \\ {ae_2}^2 \\ {ae_3}^2 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & 0 & 2 & -1 \\ 1 & -2 & 0 & 1 \\ 1 & 0 & -2 & -1 \\ 1 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} lift \\ roll \\ pitch \\ yaw \end{bmatrix}$$

Note that the solution of the linear system yields the squared values for the motor RPMs. Since the X32 soft core does not support floating point numbers, there is no implementation of the sqrt math function. We

first attempted to use an integer square root function. It is based on binary search and starts at the maximum value for the motor registers ($03FF_{16} = 1023_{10}$). The complexity is O(log n), thus at most 10*4 iterations are perfomed (since we have 4 motors). The search condition was first implemented using multiplication and then using values from a precomputed table of squares. Unfortunately, the difference was insignificant, as the control loop execution time revolved around 1ms in manual mode, for both scenarios. As a result, we decided to use the squared RPMs as commands for the motors. This saved important CPU time, while sacrificing the linear relationship between setpoints and motors (the difference is notceable only for high setpoint values).

The computed motor RPMs undergo a series of safety checks. First, negative numbers and overshoot values are clipped to $[0, 1023]$. Then, outliers (motors that drift away by more than 50% from the average) are tempered and, finally, no increments/decrements greater than 100 are allowed, to avoid motor stalling.

3.5.4 Manual, Yaw and Full Control

The difference between these three modes is the way the setpoints are computed. In *manual mode*, the joystick inputs ar mapped dirrectly to setpoints, *yaw mode* adds a P controller for the yaw rate and *full mode* adds cascaded P controllers for the roll and pitch. The P controllers compute setpoints proportional to the difference between the joystick and the sensor-determined position. Cascaded P controllers are two nested P controllers; the inner loop compares the joystick to the sensor-determined angle and the result is plugged into the outer loop, which compares it to the sensor-determined rate:

$$Setp_{yaw} = P \times (Joy - Sensor_{rate}) \qquad Setp_{roll/pitch} = P_1 \times ((P_2 \times (Joy - Sensor_{angle}) - Sensor_{rate}))$$

In addition to setting the values for the P parameters, it is possible to activate/deactivate all controllers and to set the signs of the (Joy - Sensor) error computations (to correct positive feedback) from the user interface. The values need to be brought to the same order through scaling. This is done according to the following tables:

Variable	Range
Joystick lift	[0, 126]
Joystick roll/pitch/yaw	[-126, 126]
Sensor rate	$[-20 \times 2^{11}, 20 \times 2^{11}]$
Sensor angle	$[-90 \times 2^{11}, 90 \times 2^{11}]$
P parameter	[0, 126]

$\rightarrow$

Setpoint	Range
Lift	$[0, 4 \times 1023]$
Roll	$[-1 \times 1023, 1 \times 1023]$
Pitch	$[-1 \times 1023, 1 \times 1023]$
Yaw	$[-2 \times 1023, 2 \times 1023]$

3.6 Contributions

Victor Spiridon implemented the Control and Fine-Tuning UIs (1619 LOC), the Android application (506 LOC) and the PANIC mode on the QR (124 LOC). Vlad Dumitrescu implemented the protocols on both the PC and QR (1117 LOC), the QR's state transitions, task scheduling and interrupts (659 LOC), and some protocol test programs (221 LOC). Răzvan Venter implemented initial filters in C++ (164 LOC), ported them on the QR (342 LOC) and the spectrum analysis program in Matlab (42 LOC). Matei Ţene implemented the QR's control loop, fixed-point arithmetic, motors update logic, sqrt (769 LOC), some protocol unpacking (36 LOC), the Butterworth coefficients computation in the UI (260 LOC), and tools for initial data collection and serial communication experiments (962 LOC).

4 Results

Fig. 9 shows the results of a measurements in Full Control using a prerecorded set of sensor values on the NEXYS board. The 2nd order Butterworths are activated for the yaw gyro and the accelerometers and are called in the sensors interrupt taking around 0.67 ms. The full control loop usually runs in 0.58 ms, the roll and pitch Kalman filters (called in the control loop) take around 0.35 ms, while the controller computations and updating the motors (also called in the control loop) take less the 0.2 ms. We suspect that the rare spikes are caused by the unfortunate scheduling of communication tasks while the control loop is the the middle of computation, but as the total time does not go above 0.8 ms we don't consider this a major problem. Using 1st order Butterworth improves the response a little bit with no visible impact.

All operation modes work as expected, but in full control a slow tendency of bias increase can be observed. We think it can be corrected by further tuning of the Kalman filters. Manual tests showed good response to joystick input, a very good rate controller and a decent angle controller. We managed to "fly" the QR in a, let's say, semi-controlled mode, and we think that resolving the small bias problem and, maybe, implementing a lift

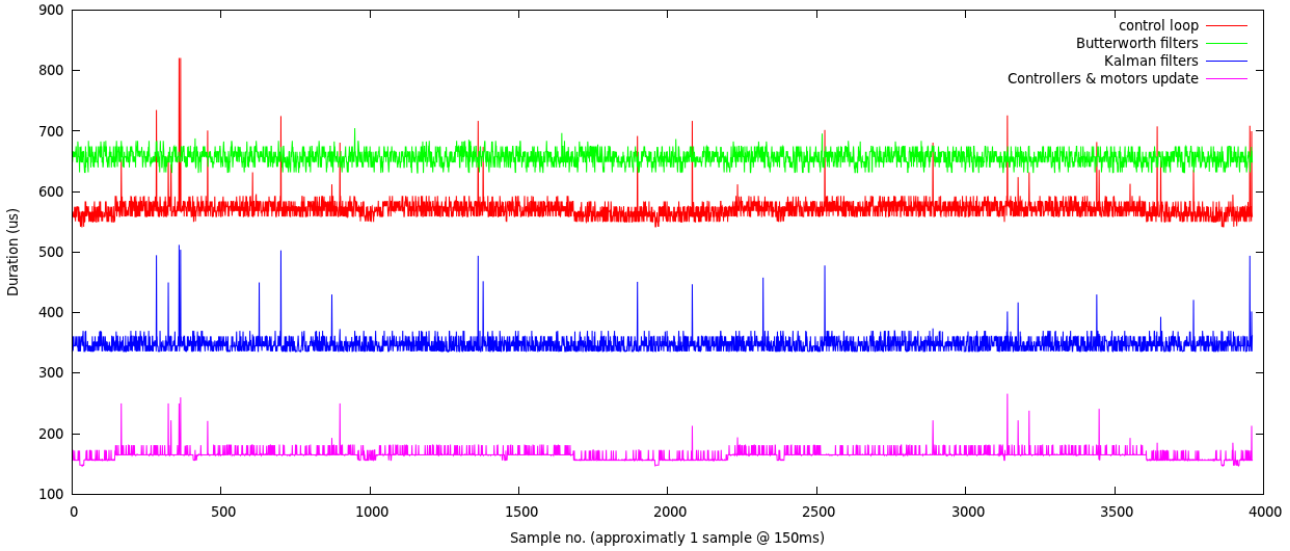

Figure 9: Measurements in Full Control mode. The 2nd order Butterworths are called from the sensors update interrupt.

controller would make it rather good. The wireless link works, but has a exhibits a very big delay because a lot of Control Protocol packets are being send and queued in the buffers.

5 Conclusions

We think that the measurements prove a good design with a very fast control loop and filtering. The scheduling algorithm does not impose a big overhead while it simplifies the code. We tested the dependability of the Control Protocol using a serial interceptor that corrupted and/or dropped bytes, thus proving it's correct functioning. The UIs are very flexible and can be used to adjust every parameter which we find useful for tuning. Also, the Android interface is a nice control interface for the user. Further work is needed to better tune the parameters and improve the wireless response by sending less control packets.

We managed to coordonate well and balance the different expertise areas of the team members. At the same time we discovered a lot of details regarding the implementation of different aspects with which we were only familiar at a theoretical level.

References

[1] IN4073 Course Web Site. `http://www.st.ewi.tudelft.nl/~gemund/Courses/In4073/index.html`.

[2] S. Woutersen, *X32 Programmer's Manual*, 2006. `http://www.st.ewi.tudelft.nl/~gemund/Courses/In4073/Resources/x32progman.pdf`.

[3] J. Hauser, SoftFloat Library. `http://www.jhauser.us/arithmetic/SoftFloat.html`.

[4] Wikipedia: Q Fixed Point Number Format. `http://en.wikipedia.org/wiki/Q_(number_format)`.

[5] Wikipedia: Butterworth Filter. `http://en.wikipedia.org/wiki/Butterworth_filter`.

[6] S. and J.R. Hollos, IIR Recursive Digital Filter Coefficient Calculation. `http://www.exstrom.com/journal/sigproc/index.html`.

[7] libev `http://software.schmorp.de/pkg/libev.html`.