# In4073 Lab Assignment 2015-2016

Koen Langendoen (course instructor)
Arjan J.C. van Gemund (founding father)

Embedded Software group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

August 2015

## 1 Introduction

The course in4073 Embedded Real-time Systems [1] aims to provide MSc students ES, CS, CE, EE, and the like, with a basic understanding of what it takes to design embedded systems (ES). In contrast to other courses on ES, especially those originating from the EE domain (traditionally the domain where ES originate from), in this course, the problem of embedded systems design focuses on using *standard, programmable* hardware components, such as microcontrollers, FPGAs, and subsequently connecting them and programming them, instead of developing, e.g., new ASICs. The rationale is that for many applications, the argument in favor of ASICs (large volumes) simply doesn't outweigh the advantages of using programmable hardware. Consequently, the course focuses on *embedded software*. The goal of the 6 ECTS credits course is not to have the student *master* the multidisciplinary skills of embedded systems engineering, but rather to have the student *understand* the basic principles and problems, develop a systems view, and to become reasonably *comfortable* with the complementary disciplines.

The project chosen for this course is to design embedded software that controls and stabilizes an unmanned aerial vehicle (UAV), more specifically, a Quad-Rotor UAV, coined "QR" for short. This application has been chosen for a number of reasons:

- the application is typical for many embedded systems, i.e., it integrates aspects from many different disciplines (mechanics, control theory, sensor and actuator electronics, signal processing, and last but not least, computer architecture and software engineering).
- The application is contemporary. Today's low-cost RC (radio controlled) model-UAVs (such as quad rotors and helicopters) are only rudimentary controlled (if at all), which implies that only highly skilled hobby pilots are capable of flying these machines (i.e., *simultaneously* controlling vertical lift, roll, pitch, and yaw[1]) without crashing within a few seconds after lift-off. While perceived as *the* true sporting challenge, many recreational users (the author included) would prefer an aerial vehicle that is much easier to handle.
- The application is typical for many air, land, and naval vehicles that require extensive embedded control software to achieve stability where humans are no longer able to perform this complicated, real-time task. Professional aerial vehicles such as helicopters, most airline and fighter jets totally rely on ES for stability. This also applies to submarines, surface ships, missiles, satellites, and space ships (future automobiles will also extensively rely on embedded software for safety-critical tasks such as steering, braking, etc., commonly denoted "X-by-wire").
- Last, but certainly not least, quad rotor vehicles are great fun, as shown by the rapidly increasing commercial interest in these toys (and this course).

Although designing embedded software that would enable a QR to autonomously hover at a given location or move to a specified location (a so-called *autopilot*) would by far exceed the scope of a 6 EECS credit course, the course project is indeed inspired by this very ambition! In fact, the project might be viewed as a *prototype study* aimed to ultimately design such a control system, where a pilot would remotely control the QR through a single joystick, up to the point where the UAV is entirely flown by software.

[1] roll, pitch, and yaw are the three angles that constitute the UAV's attitude in 3D [7]. See Appendix A.

## 2 System Setup

The ultimate embedded system would ideally be implemented in terms of one chip (a SoC, system on chip) which would easily fit within the QR's limited payload budget. Instead our prototype ES will be implemented on an *FPGA board* of half a credit card size, which provides a versatile experimentation platform for embedded system development. The system setup is shown in Figure 1. The FPGA is connected via a so-called *QR link* to a small sensor/actuator electronics interface board (IFB) that interfaces the FPGA board with the QR's actuators and sensors[2]. The ES receives its commands via a so-called *PC link* from a so-called *ground station*, which comprises a Linux PC and a joystick. The PC link can be operated in (1) wireless, and (2) tethered mode. Although wireless mode is the ultimate project goal, *tethered flight* will be most extensively used. In this mode the QR is connected to the ground system through wires. Apart from much higher communication bandwidth, this also allows for longer testing, since QR batteries are usually depleted within less than 10 minutes and rechanging them takes hours. More information on the QR, including operating instructions is found at the In4073 Resource website [8] (in particular, the QR Operations Manual).

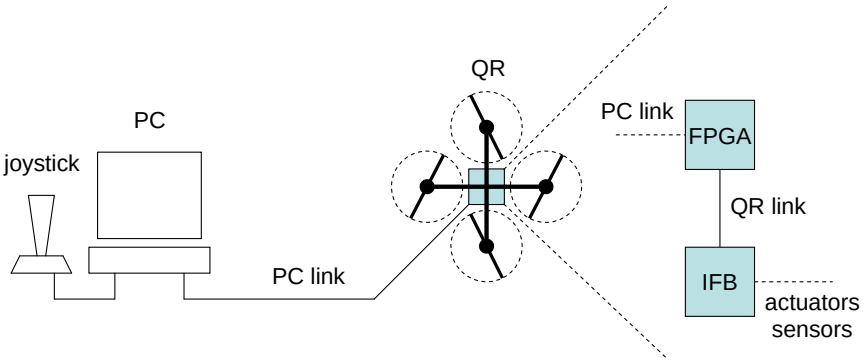


Figure 1: Hardware Setup

Rather than resorting to expensive sensors, electronics, FPGAs, etc., we explicitly choose a *low-cost approach*, which opens up a real possibility for students inspired by this course to continue working with QRs and/or ES on a relatively low budget. Hobby model QR's are relatively low-cost. As the rotors do not feature a swash plate there is no rotor pitch control. Consequently, rotor thrust must be controlled through varying rotor RPM (revolutions per minute). The four rotors control vertical lift, roll, pitch, and yaw, respectively, by varying their individual rotor RPM (see Appendix A). The sensors required to derive QR attitude (which is the minimal information for a simple autopilot application) are also low-cost. The same holds for the ES hardware approach. Rather than using some high-performance microcontroller (and costly development tool suite), we use a low-cost FPGA board ($O(100)$ Euro) featuring a 1600Kgates Xilinx Spartan-3E, that comes with *free* development software tools. In order to allow less time-critical parts of the ES to be conveniently developed in C rather than VHDL, an experimental 32 bit soft core, called X32, has been developed at our Embedded Software Lab [9], which comprises the VHDL core and associated C programming tool chain (ANSI C compiler, assembler, linker, simulator, uploader). As a TU Delft development, the X32 is obviously license-free.

The PC is a standard Linux PC, that acts as development and upload platform for the FPGA (VHDL) and the X32 soft core (C). The PC is also used as user interface / ground station, in which capacity it reads joystick and keyboard commands, transmits commands to the ES, receives telemetry data from the ES, while visualizing and/or storing the data on file for off-line inspection. The PC has serial RS232 ports to communicate to the FPGA board in wireless and tethered mode (PC link). Wireless communication is implemented using a low-cost USB transceiver operating at 868 MHz.

Of course, the low-cost approach is not without consequences. Using low-cost sensors introduces larger measurement errors (drift, disturbance from QR frame vibrations) which degrades computed attitude accuracy

[2]The interface board is, in fact, another embedded system, specifically designed for the QR's actuators and sensors. This interface is implemented using a low-cost Atmel microcontroller, that provides a 500,000 baud 8n1 serial connection to the FPGA board (QR link), translates command data from the FPGA into local engine and servo commands, converts analog sensor data into telemetry data, and transmits the telemetry data back to the FPGA board, thus closing the control loop.

and control performance. Using a low-cost FPGA (compared to a high-end FPGA with multiple hard processor cores) severely reduces the size and performance of the designs. Since the X32 occupies only 50% FPGA space, there is limited space for the additional VHDL devices (clocks, timers, serial communication devices) that make up the total microcontroller architecture. Moreover, FPGA designs typically run at much lower clock speeds compared to a high-end hard core, which results in approximately 2 MIPS for the X32. Nevertheless, experiments have shown that the low-cost setup is more than capable to perform QR control, and provides an excellent opportunity for students to get fully acquainted with embedded systems programming in a real and challenging application context, where dealing with limited-performance components is fact of life.

# 3 Assignment

The team assignment is to develop (1) a *working demonstrator* and (2) associated documentation (a *report*) of a QR control system, based on the QR available. In an uncontrolled situation, the QR, inherently unstable, will almost immediately engage in an accelerated, angular body rotation in an arbitrary direction, which typically results in a crash within seconds as an unskilled human will not be able to detect the onset of the rotation quickly enough to provide compensatory measures with the joystick. The embedded controller's purpose is to *control* 3D angular body attitude $(\varphi, \theta, \psi)$ and rotation $(p, q, r)$, as defined in Appendices A and B. This implies that the body rotation will not just uncontrollably accelerate, but will be controlled by the embedded system according to the setpoints received from the joystick (specifying `roll`, `pitch`, and `yaw`). In effect, the embedded system acts as a *stabilizer* allowing a (slow) human to manually control QR attitude using the joystick.

The lab hardware available to the team are [7] (i) a twist handle joystick, (ii) 2 Linux PCs, (iii) 2 FPGA boards, including the X32 soft core, and (iv) a QR, including power supply. The QR shall be controllable through the joystick, while a number of parameters shall be controlled from the PC user interface. The status of the system shall be visualized at the PC screen, as well as on the FPGA board (using the 8 LEDs on the QR). Whereas the hardware is provided, in principle, *all* software (excluding X32) must be *designed by the team.* Code on the Linux PC shall be programmed in C. Code on the FPGA shall be programmed in (embedded) C, and VHDL when additional X32 peripherals are needed to offload the CPU (for ES students only). A description of the required system setup, together with the signal conventions appears in Appendix B.

Although the project intentionally has some degrees of design freedom, the following requirements *must* be fulfilled. The first set of requirements relate to safe QR operation such that this valuable piece of equipment is not damaged:

**Safety** The ES must have a *safe state* (for QR and operator) that shall be reached *at all times.* In this state, the four rotor engines are *de*-energized (zero RPM). This shall be the initial state at application startup, as well as the final state (either at normal exit or abort). Consequently, joystick states other than zero throttle and neutral stick position are illegal at startup, upon which the system shall abort. Note, that the safe state is not to be implemented by merely zeroing all setpoints at the PC side as this does not guarantee that the ES (when crashed) will be able to reach this state.

**Emergency stop** It shall be possible to abort system operation at all times, upon which the system briefly enters a *panic state*, after which it goes to safe state. In the panic state rotor RPM is set at a sufficiently low rate to avoid propellor damage, yet at a sufficiently high rate to avoid the QR falling to ground like a brick causing frame damage. Panic mode should last for a few seconds until it can be assumed that touch-down has occurred, after which the QR must go into safe state. Aborting the system must be possible by pressing joystick button(s), and keyboard key(s) (see Appendix C).

**Stalling proof** Brushless engines are very efficient, but they are also prone to *stalling.* This occurs when such a sudden increase of RPM is presented to the engines such that they cannot follow. As stalling largely reduces engine power (and may result in engine damage as well), the QR will immediately become unstable and may crash. In order to avoid stalling the engine setpoint signals (`ae`) generated by the embedded control software must be kept at sufficient level during flight to avoid stalling. A practical approach is to also avoid any engine having a large RPM deviation from the average of all four engines.

**Robust to noise** As sensor data is typically noisy and may contain spurious outliers, feeding raw sensor data to the controllers may result in large variations of the four rotor control signals `ae1`, `ae2`, `ae3`, and `ae4`, possibly causing engine stalling. Sensor signal processing must therefore feature a *digital filter chain* that removes obvious outliers and filters vibration noise such that the sensor data is sufficiently stable and

reliable to be used in feedback control[3].

**Reliable communication** The PC link must be reliable, as any error in the actuator setpoints may have disastrous consequences, especially when the QR has lift-off. A wrong rotor RPM setpoint may immediately tilt the QR, causing a crash. Note that *dependability* is a key performance indicator in the project: *a major incident that is caused by improper software development (i.e., improper coding and testing) may lead to immediate disqualification.* Testing communication status at both peers may involve sending or receiving specific status messages to ascertain that both peers are operating and receiving proper data.

**Dependability** The same dependability standards apply to the embedded system itself. If, for some reason, PC link behavior is erratic (broken, disconnected) the code should immediately signal the error and go to panic mode. The same applies to the QR link. Whenever there is reason to suspect that either the QR link is unreliable, or that the QR has been shut off, an error should be flagged (also on the FPGA, as the PC, or the PC link may be inoperable) after which the code should safely shut down.

**Robustness** Last but not least, the embedded control code must be as reliable as possible. An important aspect is the proper use of fixed point arithmetic for representing and handling of sensor values, filter values, controller values, various sorts of intermediate values, parameter values, and actuator values (the X32 does not support floating-point arithmetic). On the one hand, the values must not be too small, which would cause loss of precision (resolution). On the other hand, the values must not become too large, as this may cause integer overflow, with possibly disastrous consequences. In order to guarantee the absence of erratic results the embedded code *must* include sensible exception handling, using the X32's divide-by-zero and integer overflow interrupt capabilities (see X32 manual [9]).

The next requirements relate to proper functionality of the ES. As mentioned earlier, the ES is intended to act as a controller that provides an order of stabilization to the QR attitude and rotation, such that the QR can be adequately controlled by a human. The requirements are as follows:

**PC** Apart from a VHDL and X32 C development host, the PC shall act as the user's QR *control interface.* After verifying that the joystick settings are in neutral, it shall upload the ES program image to the ES (which at that point runs a simple X32 boot monitor, featuring a program upload function), activate the ES program image (using the boot monitor's program run function), read commands from the keyboard and the joystick, send commands to the ES, read telemetry data from the ES, and visualize and store the ES data. At mission completion (or abort) the PC shall set the ES in the safe state (unless it was an FPGA button that caused the ES to return to the safe state), have the ES program terminate, which returns control to the X32 boot monitor.

**QR** The ES shall be operated in at least six control modes, enumerated 0 ("safe"), 1 ("panic"), 2 ("manual"), 3 ("calibrate"), 4 ("yaw control"), and 5 ("full control"). In modes 2, 4, and 5 the joystick and keyboard produce the control signals `lift`, `roll`, `pitch`, and `yaw` (see Appendix C for the joystick and keyboard mappings). The demonstrator shall commence in mode 0 at startup. A flight experimentation sequence always starts with selecting the proper mode, prior to ramping up engine RPM.

**Safe mode** In safe mode (mode 0) the ES sends commands to the QR to shutdown the engines. In the current version of the QR link protocol the ES might even refrain from sending anything to the QR, as the QR will shut down the engines whenever it senses loss of contact with the ES. In the panic state, the ES should ignore any command from the PC link other than the command to either move to the *manual mode* or to exit altogether.

**Panic mode** In panic mode (mode 1) the ES commands the engines to moderate RPM for a few seconds such that the QR (assumed uncontrollable) will still make a somewhat controlled landing to avoid structural damage. In the safe state, the ES should ignore any command from the PC link, and after a few seconds should autonomously enter safe mode.

**Manual mode** In manual mode (mode 2) the ES simply passes on `lift`, `roll`, `pitch`, and `yaw` commands from the joystick and/or keyboard to the QR *without* taking into account sensor feedback.

Note that the keyboard must also serve as control input device (for instance, pressing 'a' increments `lift`, pressing 'z' decreases `lift`), such that the actual commands being issued to the ES are the *sum* of the keyboard and joystick settings. This enables the keyboard to be used as static *trimming*

[3]Note that the current X32 only supports integer arithmetic [9]. Consequently, the filters must be implemented using *fixed-point* arithmetic.

device, producing a static offset, while the joystick produces the dynamic relative control component. Also see Appendix C which prescribes the key map that must be used.

**Calibration mode** In calibration mode (mode 3) the ES interprets the sensor data as applying to level attitude and zero movement. As the sensor readings contain a non-zero DC offset, even when the QR is not moving, the calibration readings thus acquired are subsequently stored during subsequent mode transition to controlled mode, and used as reference during sensor data processing in controlled mode.

**Yaw-controlled mode** In yaw-controlled mode (mode 4) the ES controls QR yaw rotation in accordance with the yaw commands received from the twist handle. In this mode, the joystick control mapping is the same as in manual mode, except for the twist handle (`yaw`), which now presents a yaw *setpoint* to the *yaw controller*. The yaw setpoint is yaw rate (cf. $r$). Neutral twist (setpoint 0) must result in zero QR yaw rate ($r = 0$). Unlike in manual mode, QR yaw should now be *independent* of rotor speed variations as much as possible, i.e., when the QR starts yawing the ES should automatically adjust the appropriate rotor thrusts `ae1` to `ae4` to counter-act the disturbance. As the yaw sensor may drift with time, an additional trim function (through keyboard keys, see Appendix C) must be supplied in order to maintain neutral twist - zero yaw relation. In this mode, the relevant controller parameter ($P$) should also be controllable from the keyboard (Appendix C).

**Full control mode** In full control mode (mode 5) the ES also controls roll and pitch angle ($\varphi, \theta$), next to yaw rate ($r$), now using all *three* controllers instead of just one. The same principles apply: neutral stick position should correspond with zero QR roll and pitch angle. As the QR has a natural tendency for rapidly accelerated roll and pitch rotation (it's an inherently unstable system), proper roll and pitch control is absolutely required in order to allow the QR to be safely controlled by a human operator. Similar to the yaw control case, *proper roll and pitch trimming prior to lift-off is crucial to safely perform roll and pitch control during lift-off* as the vehicle has a natural tendency to either pitch or roll due to rotor and weight imbalance. For the keyboard map of the roll and pitch trimming keys see Appendix C.

Note: In order to achieve optimum safety, a mode switch to modes 2, 3, 4, and 5 should only be allowed to occur under zero engine RPM conditions (`ae1` to `ae4` = 0).

**Profiling** In order for the control algorithms to work properly, the control loop time, i.e., the total latency of computing controller output, writing to the QR, reading the resulting QR response returned by the sensor filter chain, filtering the sensor values to be processed by the controller *must be in the order of 2 ms or less*. As the QR response is nearly instantaneous, most of the control loop delay is caused by the X32 code. Since the X32 runs at only 2 MIPS (it is a *soft* core) some code timing measurements will have to be performed (using the X32 $\mu$s or ms clock) as to ascertain that the entire control loop fits within the control rate budget! For example, if the entire code takes more than 1 ms, in case of a 500 Hz control frequency (2 ms), there is only 1 ms left for other tasks, such as the communication through the 115,200 baud PC link (X32 controller setpoints, PC telemetry data). Given the fact that the RS232 payload (8 bits) is embedded within a 10 bits protocol (1 start bit, 8 bits, 1 stop bit, no parity), this baud rate approximately corresponds to a 100 $\mu$s/byte communication bandwidth, i.e., only 10 bytes each 2 ms. Hence, *proper time budgeting and scheduling is a crucial step in the development of a stable and reliable QR control system!* Note that some of the X32 tasks may be moved from the C programming space to the X32 (VHDL) peripheral space (e.g., a peripheral that does all filtering, and/or control), which will significantly off-load the X32 CPU (hardware/software codesign).

**Logging** In order to analyze system performance as well as to debug coding errors the ES must have a *data logging facility*, such that all relevant signals can be visualized and can be stored to PC file for off-line analysis (possibly buffered in ES RAM first [9] and flushed at the end of the mission). Signals to be logged include ES system time, system mode, incoming joystick/keyboard data, outgoing QR actuator data, incoming sensor data, and intermediate data from the sensor signal processing chain, and relevant controller data (e.g., control parameters). Note that the signals must be time-stamped, i.e., a line of signals in the log must start with system time (microsecond resolution, see X32 capabilities [9]). In this way, system performance (system reaction speed is crucial to proper control) can be observed as well as plotted against time. Some of the most relevant data should also be real-time visible on the PC screen (max 10 Hz update rate). Minimum measurements that must be conducted (and documented) include the internal control response time of the ES (typically, the control loop time in ms), the system response time as perceived by the user (i.e., the time between joystick movement or key-press, and QR response),

and the associated QR yaw, roll, and pitch response (in terms of a time plot) in order to properly assess achieved stability performance. All measurements must adhere to the `gnuplot` / `matlab` data format.

The overall design may vary in quality and performance, which, of course, will directly translate into the course grade. Acceptable demonstrators include the following versions:

- simple actuator control through joystick and keyboard, and sensor readout at the PC, but without feedback control (will demonstrate mode 2 operation, this is the bare minimum to avoid failing the project)
- version including yaw feedback control (will demonstrate mode 2+4, without lift-off)
- version including full feedback control (will demonstrate mode 2+4+5 stabilization, with tethered lift-off).
- wireless version (will demonstrate mode 2+4+5 stabilization, with wireless lift-off). Obviously, this mode will earn the most credits, but is only allowed when tethered mode has been demonstrated to have sufficient quality/stability.

The last three versions require specific control software that is based on control algorithms for yaw, roll, and pitch. As the project does not target QR control theory, simple control algorithms are made available to the team [7].

## 3.1 System Architecture

As mentioned earlier, the main challenge of the project is to write embedded software, and thus the system (PC and FPGA) comes without much software (except for all SW development tools). In some cases however, demo SW is made available to aid in the development.

The joystick is connected to the PC via a USB link for which a driver is installed. Simple demo SW is available from [7]. The PC is connected to the FPGA through an RS232 connection (PC link). Simple demo SW is available from [7] that demonstrates a full duplex terminal program that passes keystrokes to the RS232, and concurrently passes incoming RS232 chars to the screen. At the FPGA end the PC link is operated by a VHDL component (called a UART[4]) that comes as one of the peripherals of the X32 architecture [9]. The UART is configured at 115,200 baud, 8 bits per character, no parity, 1 stop bit (8n1). The UART has a tx register for sending a character, and an rx register for receiving a character. In addition, there is a status register that can be used to monitor rx and tx buffer status. Up to 15 rx and tx characters can be buffered in the UART FIFOs. For register mapping and interrupt support, as well as for example programs see [9].

The QR link is an RS232 connection operated at 500,000 baud (again, 8 bits per character, no parity, 1 stop bit) and is supported at the FPGA end by another UART (VHDL) component that is also integrated within the X32 architecture as a peripheral. Unlike the PC link UART, which offers standard character-level I/O, the QR link UART is extended to support the frame level protocol under which all sensor and actuator values are transmitted to and from the FPGA. This VHDL component offloads the embedded C program as the high transmission rate would cause excessive load on the X32 CPU, and can be seen as an example of *hardware-software codesign*. The communication protocol is given in [8].

Apart from the above constraints, the design is totally determined by the team, i.e., system architecture, component definitions, interface definitions, etc. are specified, implemented, tested, and documented by the team. Information regarding protocols (e.g., RS232, etc.) are available on the project site [7]. Additional information is provided by the Teaching Assistants (TAs).

## 3.2 System Development and Testing

An important issue in the project is the order of system development and testing. Typically, the system is developed from joystick to QR (the actuator path), and back (the sensor path), after which the yaw, roll, and pitch controllers are sequentially implemented. All but the last stages are performed in *tethered* mode. Each phase represents a project milestone, which directly translates into team credit. **Note:** each phase must start with an architectural design that presents an overview of the approach that is taken. The design *must first be approved* by the TA before the team is cleared to start the actual implementation.

[4]Universal Asynchronous Receiver-Transmitter.

### 3.2.1 Actuator Path

First the joystick software is developed (use non-blocked event mode) and tested by visually inspecting the effect of the various stick axes. The next steps are the development of the RS232 software on the PC, and a basic controller that maps the RS232 commands to the FPGA QR interface register values. At this time, the safe mode functionality is also tested. If the design is fully tested, the system can be demonstrated using the actual QR.

### 3.2.2 Sensor Path

The next step is to implement the sensor path, involving the sensor signal processing chain including calibration capability, the extension of the ES to map the sensor signals to RS232 telemetry data, so that the sensor chain can be monitored and logged at the PC. If the sensor path is fully tested, the QR engines are enabled and the sensor path (notably the filters) are tested under simulated flying conditions (i.e., applying RPM to the engines, and gently rotating the QR frame in 3D). The obtained log file(s) must be demonstrated to the TAs, including time plots (matlab or gnuplot).

### 3.2.3 Yaw Control

The next step is to introduce yaw feedback control, where the controller is modified such that the `yaw` signal is interpreted as setpoint, which is compared to the yaw rate measurement $r$ returned by the sensor path, causing the controller to adjust `ae1` to `ae4` if a difference between `yaw` and $r$ is found (using a P algorithm). Testing includes verifying that the controller parameter ($P$) can be controlled from the keyboard, after which the QR is "connected" to the ES. By gently yawing the QR and inspecting `yaw`, $r$, and the controller-generated signals `ae1` to `ae4` via the PC screen, but without actually sending these values to the QR engines, the correct controller operation is verified. Only when the test has completed successfully, the signals `ae1` to `ae4` are connected to the QR engines and the system is cleared for the yaw control test on the running QR.

### 3.2.4 Roll/Pitch Control

The next step is to introduce roll, and pitch feedback control, where the controller is extended in conformance with mode 5 using cascaded P controllers for both angles. Again, the first test is conducted with disconnected `ae1` to `ae4` in order to protect the system. When the system is cleared for the final demonstration, the test is re-conducted with connected `ae1` to `ae4`.

### 3.2.5 Wireless Control

Only when all previous versions have been successfully demonstrated a TA can clear a team to go to *wireless* mode. In this mode the PC link is operated using the 868 MHz wireless link. The particular challenge of this version is the low bandwidth of the link, compared to the tethered version, which only permits very limited setpoint communication to the ES. This implies that ES stabilization of the QR must be high as pilot intervention capabilities are very limited.

### 3.2.6 Methodology

The approach towards development and testing must be professional, rather than student-wise. Unlike a highly "student-like", iterative try-compile-test loop, each component must be developed and tested in isolation, *preferably outside of lab hours*, as to optimize the probability that during lab integration (where time is *extremely limited*) the system works. This especially applies to tests that involve the QR, as QR availability is limited to lab hours. The situation where there is only limited access to the embedding environment is akin to reality, where the vast majority of development activities has to be performed under simulation conditions, without access to the actual system (e.g., wafer scanner, vehicle) because it either is not yet available, or there are only a few, and/or testing time is simply too expensive.

## 3.3 Reporting

The team report, preferably typeset in Latex, should contain the following

- Title, team id, team members and student numbers, date
- Abstract (10 lines, specific approach and results)

- 1. Introduction (including problem statement)
- 2. Architecture (all software components + interfaces)
- 3. Implementation (how you did it, and *who did what*)
- 4. Experimental results (list capabilities of your demonstrator)
- 5. Conclusion (evaluate design, team results, individual performance, learning experience)
- Appendices (all component *interfaces*, component *code* for those components you wish to feature)

Specific items that *must* be included are a functional block diagram of the overall system architecture, the size of the C code, the control speed of the system (control frequency, the latencies of the various blocks within the control loop), and the individual contributions of each team member (no specification = no contribution). The report should be complete but should also be as minimal as possible. In any case, the report *must not exceed 10 pages* (A4, 11 point), *including* figures and appendices. Reports that exceed this limit are NOT taken into consideration (i.e., desk-rejected)!

## 3.4 Course Grading

Each team executes the same assignment and is therefore involved in a competition to achieve the best result. The course grade is directly dependent on the project result in terms of absolute demonstrator and report quality as well as relative team ranking. Next to team grading, *each team member is individually graded depending on the member's involvement and contribution to the team result.* Consequently, a team member grade can differ from the team grade.

A better result earns more credits. A result is positively influenced by a good design and good documentation (which presents the design and experimental results). A good design will achieve good stability, use good programming techniques, have bug-free software, demonstrates a modular approach where each module is tested before system-level integration, and therefore adhere to the "first-time-right" principle (with respect to connecting to the real QR).

On the debit side, to avoid lack of progress a design may sometimes require excessive assistance from the Teaching Assistants (consultancy minutes), which will cost credits.

## 3.5 Lab Order

In the absence of the course instructor, the TAs have authority over all lab proceedings. This implies that each team member is held to obey their instructions. Students that fail to do so will be expelled from the course (i.e., receive *no* grade).

Safety (systems dependability) is an important aspect of the code development process, not only to protect the QR equipment, but also to protect the students against unanticipated reactions of the QR. The RPM of the rotors can lead to injury, e.g., as a result of broken rotor parts flying around as a result of a malfunction or crash. Therefore, students who are in the immediate vicinity of the QR *must* stay out of range and wear goggles.

As mentioned at the course web page, lab attendance by all team members is *mandatory* for reasons of the importance of the lab in the course context, and also because of team solidarity. Hence, the TAs will at least perform a presence scan at the start and at the end of each lab session. Any team member who's aggregate AWOL (absent without leave) is more than 30 minutes will be automatically *expelled* from the course.

Although it is logical that team members will specialize to some extent as a result of the team task partitioning, it is crucial that each member fully understands all general concepts of the entire mission. Consequently, *each team member is expected to be able to explain to the TAs what each of his/her team members is doing, how it is done, and why it is done.* The TAs will regularly monitor each team member's performance. Team members that display an obvious lack of interest or understanding of team operations risk having their individual grades being lowered or even risk being expelled from the course. As a special check on member performance, and even fraud, each member must *author* each C *function* he/she has written. Note that C functions *must be single-authored*. Functions that have multiple authors simply implies that those functions are too big, which is unacceptable. The entire source tree (PC and X32) must be made available to the TAs at all times.

Fraud (as well as aiding to fraud) is a serious offense and will always lead to (1) being expelled from the course and (2) being reported to the EEMCS Examination Board. As part of an active anti-fraud policy, all code must be submitted as part of the demonstrator, and will be subject to extensive cross-referencing in order

to hunt down fraud cases. It is NOT allowed to reuse ANY In4073 computer code or report text from anyone else, including code or text produced by one's self during an earlier In4073 course edition (fraud), nor to make In4073 code or text available to any other student attending In4073 (aiding to fraud). An obvious exception is the code that has been made available on the In4073 Resource web site. NOTE: an excuse that one "didn't fully understand the rules and regulations on student fraud and plagiary at the Delft Faculty of EEMCS" will be interpreted as an insult to the intelligence of the course instructor and is NOT acceptable. In case of doubt one is advised to first send a query to the course instructor *before acting*.

Finally, note that the lab sessions only provide a forum where teams can test their designs using the actual QR hardware. However, these sessions are *not* enough to successfully perform the assignment. An important part of the team work has to be performed *outside lab hours*, involving tasks such as having team meetings, designing the software, preparing the lab experiments, and preparing (sections of) the report. In order to enable (limited) experimentation outside lab hours, one or two FPGA kits per team can be made available outside lab hours, once a EUR 100 deposit has been made. The deposit is reimbursed once the kit has been returned and it is established that the kit is *fully functioning*. Failing to return a kit is subject to sanction, including being expelled.

## Acknowledgments

The founding father of this course is Arjan van Gemund, who is now enjoying the fine life of a retired professor. He conceived the lab and guided numerous people in implementing essential building blocks (HW and SW) for several generations of quadcopters, eventually arriving at the QRs in use today.

Marc de Hoop has developed the first QR interface electronics as part of his MS thesis work on the X-UFO (the QR used up to 2008-2009). Sijmen Woutersen (a previous TA) has been responsible for setting up the Xilinx tool chain under Linux, and developing the X32 tool chain, respectively (the X32 has been Sijmen's MS thesis work). Also thanks to Mark Dufour, who has ported the RTOS uC/OS-II to the X32, and who has developed the X32 debugger. Many thanks go to 2007-2008 TAs Tom Janssen and Michel Wilson. Tom for solving the many problems that arose throughout the last 4 years, in which we evolved from Picolo Micro heli, via the TREX 450 heli, to the 2007-2009 X-UFO QR, and Michel for has share during the last 2 years, researching some JTAG cable related parallel port driver and USB-to-RS232 cable driver issues under Linux, and for porting the X32 to the new NEXYS-II FPGA board (including the usual hassle with memory controllers). The 2008-2009 TAs Widita Budhysusanto and Tiemen Schreuder have developed a simple, but effective controller that already achieves pretty good roll and pitch stabilization (roll and pitch are much more difficult than yaw). Our 2009-2010 TA Matias Escudero Martinez ported the X32 to the latest TE0300 FPGA board, including the design of a complicated memory controller for the DDR SDRAM that comes with this board. Finally, 2009-2011 TA Shekhar Gupta has verified the improved control approach using Kalman filtering. In addition, 2010-2011 TA Imran Ashraf has extended the QR peripheral to send all four engine updates in VHDL instead of C to further offload the X32.

## References

[1] In4073 Course Web Site. `http://www.st.ewi.tudelft.nl/~koen/in4073/`.

[2] B. Etkin, L.D. Reid, *Dynamics of Flight, Stability and Control*, 3rd Ed., 1996, Wiley.

[3] Digilent Nexys2 Board Reference Manual, `http://www.st.ewi.tudelft.nl/~koen/in4073/Resources/`.

[4] Gnuplot Tutorial. `http://www.duke.edu/~hpgavin/gnuplot.html`.

[5] Technical Report, Faculty of Aerospace Engineering, TUD, 2007. `http://insight.oli.tudelft.nl/final.pdf`.

[6] QR Simulator. A.J.C. van Gemund. `http://www.st.ewi.tudelft.nl/~koen/in4073/Resources/`.

[7] In4073 Resources `http://www.st.ewi.tudelft.nl/~koen/in4073/Resources/`.

[8] QR Manual 2009-2010. A.J.C. van Gemund. `http://www.st.ewi.tudelft.nl/~koen/in4073/Resources/qr.pdf`.

[9] X32 Web Site (including Programmer's Manual). `http://x32.ewi.tudelft.nl/`.

# A QR Theory of Operation

In Figure 2 we describe the variables that are standard terminology in describing aerial vehicles location and attitude (NASA airplane standard). We use two coordinate frames, the earth frame and the (QR) body frame. When the QR (body frame) is fully 3D aligned with the earth frame, both $x$ axes are pointing in the direction of the QR's flight direction. The $y$ axes are pointing to the right, while the $z$ axes are pointing *downward*. The

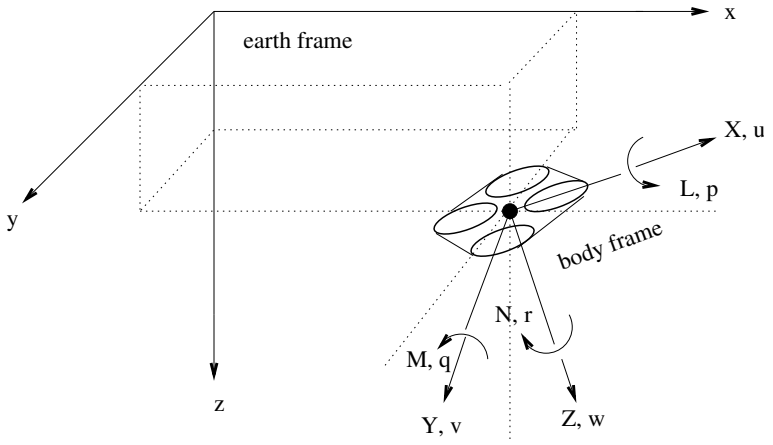


Figure 2: Standard QR variables

variables are defined as follows:

- $x, y, z$: coordinates of the QR CG (center of gravity) relative to the *earth frame* [m]
- $\varphi, \theta, \psi$: QR attitude (body frame attitude) relative to the *earth frame* (Euler angles) [rad]
- $X, Y, Z$: forces on the QR relative to the *body frame* [N]
- $L, M, N$: moments on the QR relative to the *body frame* [N m]
- $u, v, w$: QR velocities relative to the *body frame* [m $s^{-1}$]
- $p, q, r$: QR angular velocities relative to the *body frame* [rad $s^{-1}$]

The forces $X, Y, Z$ result in (3D) acceleration of the vehicle, while the moments $L, M, N$ result in rotation (change in roll, pitch, yaw). In hover, $Z$ determines altitude (to be controlled by `lift`), while $L, M, N$ need to be controlled by `roll`, `pitch`, and `yaw`, as explained later on.

The Euler angles $\varphi$, $\theta$, and $\psi$ are not included in Fig. 2 as their graphical definition is less trivial. They describe the attitude of the body frame relative to the earth frame, and are defined in terms of three successive rotations of the *body frame* by the angles $\psi, \theta, \varphi$, respectively[5]. Before applying the rotations the body frame is fully aligned with the earth frame. First, the body frame is rotated around the $z$ axis (body frame, earth frame) by an angle $\psi$. Next, the body frame is rotated around the $y$ axis (body frame) by an angle $\theta$. Finally, the body frame is rotated around the $x$ axis (body frame) by an angle $\varphi$. Note, that the order of rotations $(\psi, \theta, \varphi)$ is significant as different rotation orders will produce a different final body frame attitude. The Euler angle definition is shown in Figure 3, where the index $(i)$ denotes the various body frames in the course of rotation (0 initial body frame, 3 final body frame).

The forces $X, Y, Z$ and moments $L, M, N$ are applied externally through gravity, and the four rotors. The above forces and moments are given by

$$\begin{aligned} X &= 0 \\ Y &= 0 \\ Z &= -b(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \end{aligned}$$

[5]In QR and helicopter terminology, the three rotations are known as *yaw*, *pitch*, and *roll*, respectively, and are referred to as *heading*, *attitude*, and *bank* in airplane terminology.

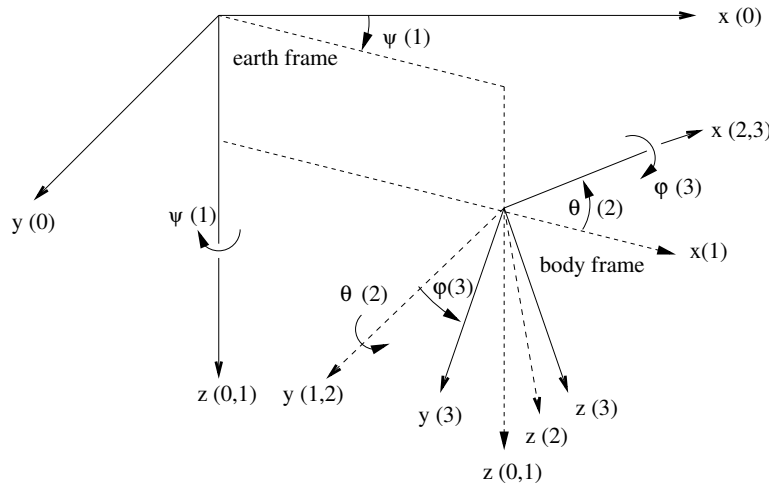


Figure 3: Body frame attitude using Euler angles

$$\begin{aligned} L &= b(\omega_4^2 - \omega_2^2) \\ M &= b(\omega_1^2 - \omega_3^2) \\ N &= d(\omega_2^2 + \omega_4^2 - \omega_1^2 - \omega_3^2) \end{aligned}$$

where $\omega_1, \ldots \omega_4$ denote rotor RPM, and $b$ and $d$ are QR-specific constants (taken from [5]). The above equations directly follow from the rotational direction of the rotors as shown in Fig. 4. The rotor RPM $\omega_i$ is approximately

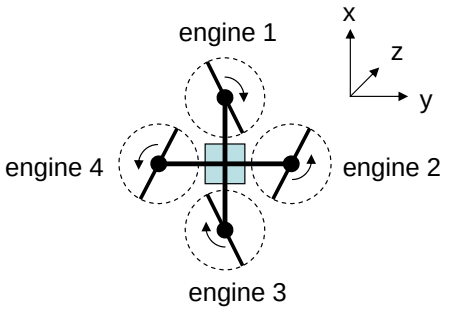


Figure 4: QR rotor layout (top view)

proportional to the voltage applied to the engines, which, in turn, is proportional to the duty cycle $c_i$ of the PWM signals generated by the QR interface electronics, which are controlled by `aei`. As a result, the above equations can also be directly expressed in terms of the actuator signals `ae1` to `ae4` according to

$$\begin{aligned} X &= 0 \\ Y &= 0 \\ Z &= -b'(ae_1^2 + ae_2^2 + ae_3^2 + ae_4^2) \\ L &= b'(ae_4^2 - ae_2^2) \\ M &= b'(ae_1^2 - ae_3^2) \\ N &= d'(ae_2^2 + ae_4^2 - ae_1^2 - ae_3^2) \end{aligned}$$

where $b'$ and $d'$ are QR-specific constants. Note, that in order to compute the `aei` to produce a desired lift (through $Z$), roll (through $L$), pitch (through $M$), and/or yaw (through $N$), the above system of equations must, of course, be inverted. An example solution is given in the QR simulator code [6].

The other variables $u, v, w$ and $p, q, r$ are governed by the standard dynamical and kinematic equations that govern movement and rotation of any body in space [2], which are also implemented in the QR simulator [6].

# B Signal Definitions

In the interest of clarity and standardization, in following we define some of the most important signal names as to be used in the project. We refer to the generic system circuit shown in Figure 5. The four most important

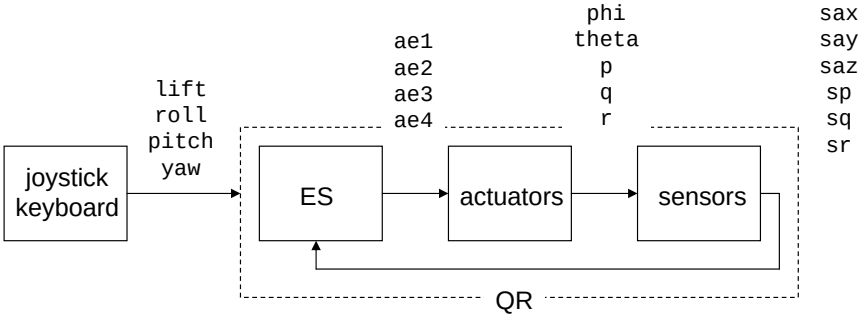


Figure 5: Standard system variables

signals originating from joystick or keyboard (see Appendix C) are called

- `lift` (engine RPM, keyboard: `a/z`, joystick: throttle)
- `roll` (roll, keyboard: `left/right arrows`, joystick: handle left/right (x))
- `pitch` (pitch, keyboard: `up/down` arrows, joystick: handle forward/backward (y))
- `yaw` (yaw, keyboard: `q/w`, joystick: twist handle)

The actuator (servo) signals to be sent to the QR electronics are called

- `ae1` (rotor 1 RPM)
- `ae2` (rotor 2 RPM)
- `ae3` (rotor 3 RPM)
- `ae4` (rotor 4 RPM)

The sensor signals received from the QR electronics are called

- `sp` ($p$ angular rate gyro sensor output)
- `sq` ($q$ angular rate gyro sensor output)
- `sr` ($r$ angular rate gyro sensor output)
- `sax` ($a_x$ x-axis accelerometer sensor output)
- `say` ($a_y$ y-axis accelerometer sensor output)
- `saz` ($a_z$ z-axis accelerometer sensor output)

Note that the sensor signals `sax, sp` etc. do *not* equal the actual QR angle and angular velocity $\theta, p$ etc. as defined earlier. Just like other QR state variables such as $x, y, z, u, v, w$, etc., they can*not* be directly measured (unfortunately), which is why the sensors are there in the first place. Rather, the sensor signals are an (approximate!) indication of the real QR state, and proper determination (recovery) of this state are an important part of the embedded system. For simulation purposes [6], where the true QR state variables are obviously available (i.e., simulated), the QR state variables are called

- `x, y, z` $(x, y, z)$
- `phi, theta, psi` $(\varphi, \theta, \psi)$
- `X, Y, Z` $(X, Y, Z)$
- `L, M, N` $(L, M, N)$
- `u, v, w` $(u, v, w)$
- `p, q, r` $(p, q, r)$

which implies that these names must be used when referring to these signals in the C code.

# C Interface Requirements

The following prescribes the minimum interface requirements. Apart from these requirements, appropriate on-screen feedback on QR and embedded systems performance earns credits.

## C.1 Joystick Map

- joystick throttle up/dn: `lift` up/down
- joystick left/right: `roll` up/down
- joystick forward/backward: `pitch` down/up
- joystick twist clockwise/counter-clockwise: `yaw` up/down
- joystick fire button: abort/exit

The other buttons can be used at one's own discretion.

## C.2 Keyboard Map

- `ESC`: abort / exit
- `0` mode 0, `1` mode 1, etc. (random access)
- `a/z`: `lift` up/down
- left/right arrow: `roll` up/down
- up/down arrow: `pitch` down/up (cf. stick)
- `q/w`: `yaw` down/up
- `u/j`: yaw control `P` up/down
- `i/k`: roll/pitch control `P1` up/down
- `o/l`: roll/pitch control `P2` up/down

The mode 5 P1 and P2 control parameters simultaneously apply to both the pitch and roll cascaded P controllers, which are identical. The other keys can be used at own discretion.

## C.3 FPGA Map

- `led[0]`: blink at a 1 s rate to indicate X32 program is running properly
- `led[1]`: on when PC link is properly functioning
- `led[2]`: on when QR link is properly functioning

The remaining LEDs can be used at one's own discretion (typically used for status/debugging).