ENCE361-19S1: Helicopter Rig Controller Project

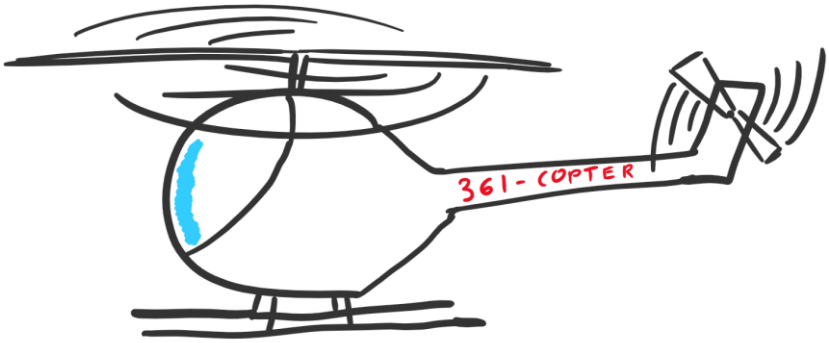
Friday Morning, Group 7

By

Jesse Sheehan 53366509

Will Cowper 81163265

Manu Hamblyn 95140975



Contents

1 Introduction 2

2 System Design 2

2.1 Key design considerations 2

2.2 Kernel 3

2.3 Inter-Task Communication 5

2.4 PID control 5

2.5 Special features 7

2.6 Potential Improvements 7

3 Conclusion 7

1 Introduction

The goal of this project was, as a group of three students, to design and develop the embedded software system to control a rig mounted a remote model helicopter. The required inputs, outputs, expected system behaviour, and hardware were given in a project specification.

Project deliverables included two milestones prior to demonstration, then final code and report, hence the group structured the software by task specific modules. This made debugging and extension easier, plus simplified project progression from milestones to final delivery.

Key learning points included:

- Implementing code that manages shared data access.
- Understanding and coding ADC sampling and averaging.
- Dealing with specification ambiguity (yaw reference) by repeated testing.
- Understanding system timing, task prioritisation, frequency and kernel implementation.
- Producing a control system that can accommodate significant differences in rig response.
- Coding a PID control that maintains system stability and does not cause physical damage to rigs.

2 System Design

The software is modular, with each module providing a set of related tasks thus providing high cohesion within modules and low coupling between modules. This allows easy refactoring of code as the implementation can be changed without affecting the contract stipulated by the header files.

2.1 Key design considerations.

- **System Clock:** A 40MHz system clock is defined in clock.c. This clock speed was chosen to allow enough time to process all the kernel tasks. Both the OLED display and UART tasks consume significant processing time hence the higher clock rate provides greater flexibility to the kernel.
- **SysTick Timer:** The SysTick controls kernel timing, see kernel.c. The required kernel is initialised with an update frequency of 40 kHz, this is the frequency that the kernel tick counter is incremented. The higher this frequency, the more accurate the kernel can be with its task profiling and scheduling.
- **ADC Buffer:** A circular buffer size of 16 was used to minimise the time to average the sampled altitude values, and to limit the reduction in effective sampling frequency caused by averaging. Circular buffers were also used to determine if the altitude and yaw had “settled” around a specific value. See altitude.c.
- **ADC sampling rate:** The ADC sampling rate needs to be faster than the (PID) controller rate. Since the (PID) controller rate is 30 Hz and circular buffer size is 16, a chosen ADC sampling rate of 512 Hz gives an effective rate of 32 Hz.
- **ADC Percentage Conversion:** A voltage change of -0.8 V corresponds to altitude change of 100%, thus: the maximum ADC is 3.3V. With a 12-bit ADC, this gives an altitude difference of 4095 * 0.8 / 3.3 = 993 bits. The altitude delta is used to convert the averaged buffer value to a percentage.
- **Yaw Reference:** Setting the yaw reference input to rising edge, weak pull up gave the reliable yaw reference detection. Note: The project description was ambiguous in its specification of the yaw reference. See yaw.c.
- **Yaw:** Yaw uses a bearing format of 0 to 360 (rather than $\pm$180) degrees.
- **Slot Count To Degrees Conversion:** The maximum number of slots was defined as 448 (112 total slots over four phases), thus the rotation in degrees is the current slot count multiplied by 360 divided by the maximum slots. Line 298 in yaw.c shows `g_slot_count * 360.0f / YAW_MAX_SLOT_COUNT.`
- **Quadrature State Machine:** A state machine uses the previous yaw state and current yaw state to determine the direction of rotation. Implementation is derived from:

https://cdn.sparkfun.com/datasheets/Robotics/How%20to%20use%20a%20quadrature%20encoder.pdf

- **PWM Frequency:** In pwm.c 200 Hz was chosen since it was an even integer clock division of the system clock (40 MHz) and using the 300 Hz specification maximum seemed to cause the rigs to shut down more often.
- **Flight Mode:** A finite state machine (FSM) controls the system modes and responses. States are advanced by the "select" switch as well other conditions such as yaw and altitude. See flight_mode.c.

2.2 Kernel

The kernel is comprised of an array of tasks that are run in a round robin fashion depending on their execution frequency and priority. Tasks are added to the kernel once during the system initialisation sequence. A task consists of a function pointer, a frequency and a priority as well as some other internal fields required to schedule the tasks properly. The kernel is initialised with a constant frequency; this is the maximum frequency at which the kernel can schedule tasks. All tasks in the project (with the exception of interrupt driven and initialisation tasks) are carried out by the kernel.

Only polled tasks can be scheduled by the kernel, ISRs must run separately. Tables 2.2.1 and 2.2.2 below show the task timings.

2.2.1 Kernel Task Information

Task Name	Duration (µs)†	Period (ms)	Frequency (Hz)	Priority
Update Altitude ADC	5.00	3.91	256	1
Update Altitude Mean	72.66	3.91	256	2
Update Altitude Settling	5.00	100.00	10	10
Update Altitude Control	15.43	333.33	30	5
Update Input	25.07	N/A	0*	50
Update Display	9020.45	1000.00	1	100
Send UART Flight Data	9959.35	250.00	4	100
Update Flight Mode	24.91	50.00	20	10
Update Yaw Settling	5.07	100.00	10	10
Update Yaw Control	17.98	333.33	30	5

Table 2.2.1: Kernel task timing

* = Run every time the kernel is updated.
† = Duration averaged via the kernel's profiling capability.

2.2.2 Interrupt Task Information

Task Name	Duration (Cycles)	Duration (µs)^
ADC sample ISR	97	2.43
Yaw ref. handler ISR	48	1.20
Yaw handler ISR	26	0.65
SysTick ISR	16	0.40

Table 2.2.2: Interrupt task timing

^ = Duration in microseconds calculated from the 40MHz system clock speed.

2.2.3 Kernel Design

The kernel also provides some temporal context to each task, namely the amount of time (in microseconds) since the task was last run, as well as the task duration (also in microseconds) since the last time it was run. To achieve (relatively) accurate timing, the kernel registers an ISR for the System Tick and keeps track of the time (in ticks) since each task was last run. The system tick handler is called with

the same frequency as what the kernel was initialised with, therefore it is important to select a high enough kernel frequency so that tasks can be run as often as needed.

A task is initialised with a name, a function pointer, a frequency and a priority. If a task is initialised with a frequency of 0 Hz then it is run as fast as is possible. Note that this is usually not desirable and will result in the task being run irregularly. The kernel will constantly check if each task in the task array is ready to be run, when it finds a task that is ready it will execute it, update some metadata then continue to the next task. Profiling of kernel tasks is also performed by the kernel (Figure 2.3.1). Analysis of the kernel task profiles revealed that the kernel tasks use on average 7% of the CPU processing time, with the remaining 93% being used for ISRs and kernel processing. Of all the kernel tasks, the most temporally expensive ones were sending UART data (3.96% of all CPU time), altitude average calculations (1.85%), and the OLED display functionality (0.90%).

Because the system clock is running at 40 MHz and our kernel is running at 40 kHz, there is an effective timing resolution of 2.5 µs. This means that it may not be possible to for the kernel to profile tasks that take less than 2.5 µs (100 clock cycles) to complete.

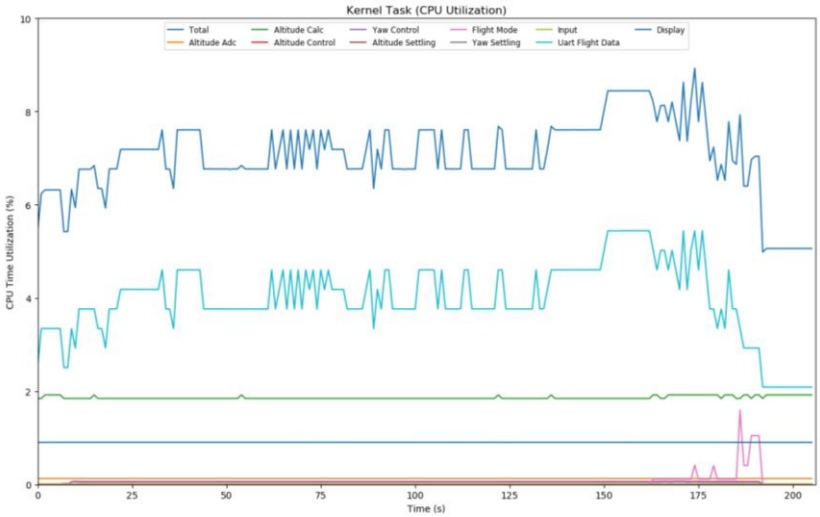


Figure 2.3.1: The percentage of CPU time consumed by the kernel tasks. Note that the CPU time consumed by ISRs and the kernel itself are not included.

Because time slicing is not used, only one task is actually being run at a time (with the exception of the ISRs). This allows us to make reasonable assumptions about the state of the system at any given time. However, this also means that a long-running task can prevent other tasks from executing when they should, slowing the system down or even halting it completely. The amount of time between how often the task should be executed and how often is actually executed is called the time error (Figure 2.3.2.3.1). The lower the time error, the more responsive the system becomes. Note that in Figure 2.3.1, the time error increases when the helicopter is in its landing state. This is partly due to the flight mode update task consuming more CPU time causing the altitude calculation task to be delayed.

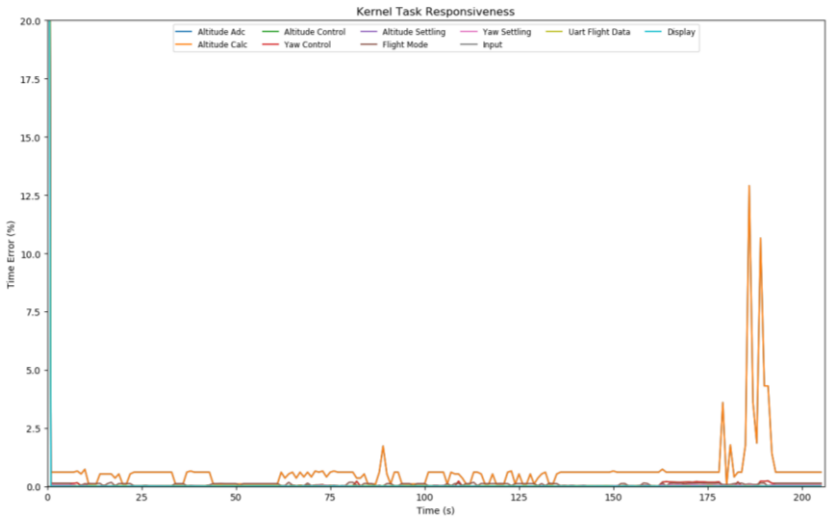


Figure 2.3.2: The time error of each task as a percentage of the task's period.

2.3 Inter-Task Communication

Each module manages its own global static variables. If the variable is written to inside an ISR, it also has the volatile modifier so the optimiser does not make any assumptions about it. Each volatile variable also has a mutex associated with it to reduce the likelihood of a volatile variable being changed by an ISR while a kernel task is reading or writing to it. This does not completely solve the shared access problem because our kernel does not allow for deadlock detection, but it does make it extremely unlikely.

Each module provides getter and setter methods where applicable, to allow other modules to get and set global variables. This allows us to check the bounds on the values being set and to make use of the mutexes in some cases.

2.4 PID control

Stability of the helicopter is maintained by independent PID controller functions for both the tail and main rotor motors. Earlier iterations of the PID control employed a controllable coupling between the main and tail rotor, however this had the effect of slowing down the response during some simultaneous altitude and yaw manoeuvres. For example, in some landing scenarios it would be necessary to reduce main rotor duty while increasing tail rotor duty to reach the desired set point promptly. A decision was therefore made that the tail and main rotor control systems should work completely independent of one another.

Data relevant for the calculation of new gains such as K_p, K_i and K_d as well as the previous and accumulated errors are stored in a global static struct for each controller. When a PID function is called, two getter functions are used to retrieve the current position and desired state of interest. From these two states an error is computed then processed to calculate the new duty cycle for the motor. This new calculated duty cycle is then passed to a PWM setter function to be realised.

2.4.1 Gain selection

Our gains were determined through an extensive iterative process. To ensure initial stability the yaw PID was disabled and the tail motor duty coupled directly to the main rotor. From here several gains were tested for the altitude PID until satisfactory response was achieved. The altitude PID includes a large (25%) positive offset to reduce the dependence on the integral error for take-off. With a working altitude PID controller the coupling of the tail rotor was then gradually relaxed, and gains introduced to allow for proper yaw control. Finally, the tail rotor coupling was removed entirely once it was proven the yaw PID controller was capable of keeping the system stable and providing suitable response to inputs.

2.4.2 Value clamping

Clamps were used liberally throughout both PID controllers. Implementing clamps allowed us to use larger K parameters to produce a better response to typical single button press scenarios, while keeping the system stable for larger offset errors from rapid successive button presses. One of the most effective implementations of clamps was in the integral error portion of both PID controllers. By clamping the rate at which the integral error could change, a higher K_i value could be used to dramatically reduce the settling time for small errors without causing instability with larger errors. Overshoot for larger errors was also reduced by clamping the lower and upper bounds of the integral error relative to the minimum and maximum duty cycle respectively.

2.4.3 Altitude and yaw control response

Figure 3.3.1 shows the altitude response to a 10% step increase input and figure 3.3.2 shows the yaw response to a 15^o step change input. In both cases rise time is less than 1.5 seconds, overshoot is limited, settling time is less than three point five seconds, and there is zero steady state error.

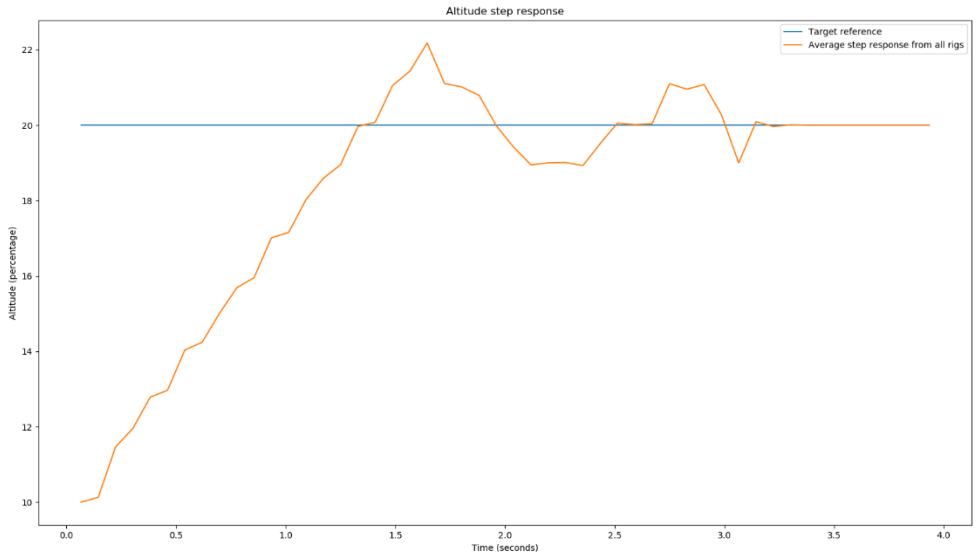


Figure 3.3.1: Altitude response to 10% change input

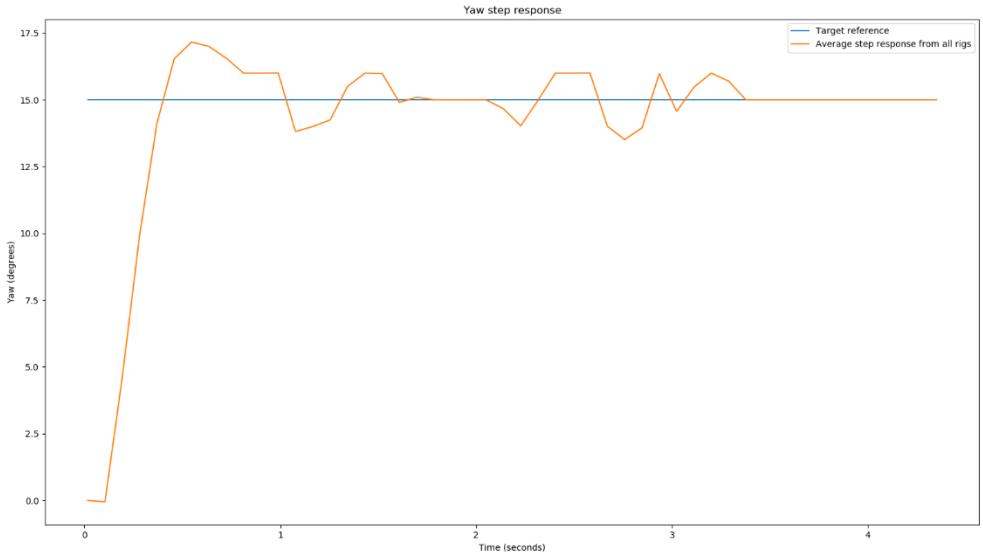


Figure 3.3.2: Yaw response to 15-degree change input

2.5 Special features

2.5.1 Landing Sequence

When the helicopter is in the IN_FLIGHT state and the “select” button/switch is initiated, the helicopter is put into landing mode. The following steps take place in order to enter the LANDED state:

- The helicopter aligns to reference yaw value (0 degrees).
- The helicopter is lowered to 10% of full height.
- Once settled at yaw reference and at 10% height for 0.5 seconds, it is lowered to 0% height.
- Once it reaches 0% height, the PID controllers are disabled, and rotors are turned off.

2.5.2 Kernel Task Prioritisation

The kernel priorities its tasks via each task’s priority value. The lower the priority value, the higher priority value it has. If two tasks have the same priority value, then the one that is added to the list last has precedence. Internally this is done with the qsort function from stdlib.h.

2.5.3 UART Data Transmission/Reception

The UART module is responsible for transmitting the flight data to the outside world. It is also responsible for transmitting information about kernel tasks if it is configured to do so. Previously, the UART was configured to also receive new control system gains from a computer that it was directly connected to, this would reduce the amount of time spent fine-tuning the gains. However, this feature was removed once it was discovered that the heli-rig system had only one-way serial transmission.

2.6 Potential Improvements

2.6.1 PID Control

The PID controllers can withstand some variation between rigs however a number of rigs had drastically different characteristics that would require significant altering of parameters. Implementing a calibration routine to calculate rig variables such as take-off duty cycle and yaw friction would have made our PID even more tolerant of different rig variations.

2.6.2 Kernel Implementation

The kernel is a very basic round-robin type and lacks many useful features but keeps shared data access simple. It could be improved by implementing time-slicing which would allow long-running tasks to be put to sleep or stopped. This would likely take much effort and would introduce a slew of shared data access problems. However, if these problems could be solved, the system would be much more responsive.

3 Conclusion

The project was undertaken to control a helicopter using an embedded system. Design challenges included ADC, PWM, task scheduling, UART, control systems and many others. These were solved via careful planning of modules, allowing separation of concerns to drive the overall software architecture. Overall, the helicopter system design and implementation worked well on most helicopter rigs and, with some fine tuning of the control system gains, could run perfectly on all.

Specific aspects that worked well included:

- Applying a modular approach to task implementation.
- Controlling access to shared variables.
- The ability to implement test and progress from test rig, to emulator, to full helicopter rig.

Specific aspects that could be improved:

- Helicopter rig access and variability in response proved challenging.
- Yaw reference.