Tutorial session: **Tuesday**

Group number: **14**

Zach Preston - zsp10

Hank Wu - jhw83

Jason Ui - jui11

Randipa Gunathilake - wrg20

ENCE464
Embedded Software

2021

University of Canterbury

Tiva C-Series HeliRig FreeRTOS Controller

Group 14

Table of Contents

1. Introduction

1.1 Background

This report documents the design, performance results and review of a closed-loop control system for the University of Canterbury HeliRig simulated-helicopter. As seen in Figure 1, the HeliSim simulates the behaviour of a two-rotor helicopter across a variety of variable physical parameters. General purpose input/output (GPIO) from a connected Nucleo STM32F072 development board stream the altitude and heading (yaw) of the helicopter to a Tiva C-Series TM4C123GXL development board.

This project focused on the development of a closed-loop control program developed on the Tiva development board. The objective of this program was to control the altitude and heading of the HeliRig according to inputs made by a user. In addition to the functional requirements outlined in Section 1.2, the HeliRig control system was developed to showcase high-level programming design patterns in concurrency, thread-safe operation, and SOLID design.

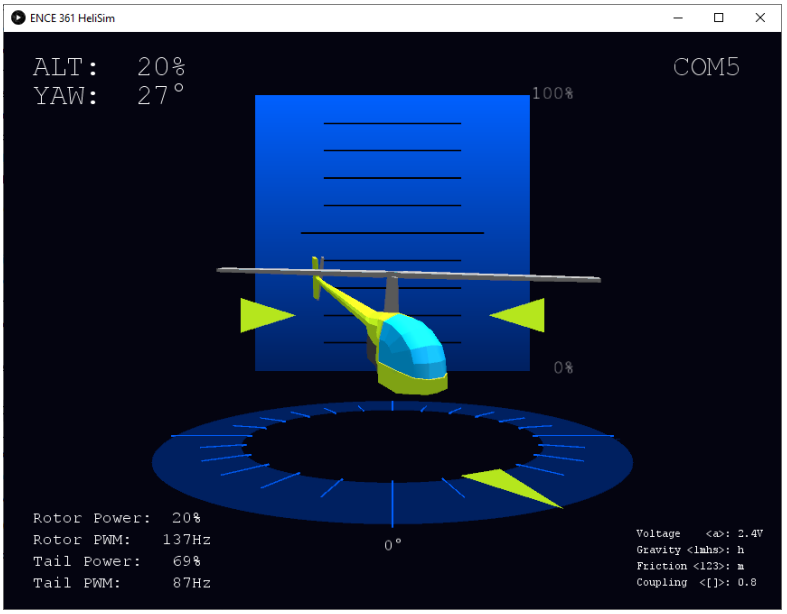

Figure 1: Screenshot of the HeliSim helicopter emulator program.

1.2 Success Criterion

The HeliRig control system would be successful if it meets the following criteria:

- The altitude of the HeliSim may be read from the associated analogue pin by the Tiva board.
- The heading (yaw) of the HeliSim may be read from the associated GPIO pins on the Tiva board.
- A user interface will allow users to control the HeliSim using the four on-board buttons. The Orbit OLED screen should also provide feedback on the current state of the helicopter rig.
- A control algorithm should be developed that allows for altitude or yaw control of the HeliSim.
- The completed program must use the FreeRTOS operating system to implement user defined tasks.

2. Software Design

2.1 Program Architecture

2.1.1 Repository Hierarchy

The HeliRig software architecture was designed using a Model View Controller (MVC) framework. This framework was used to identify and categorise the modules that would be required to accomplish the program objectives. As seen in Figure 2, the program hierarchy was modularised into three key subdirectories: app, drivers and utilities. The *app* directory contained all of the program tasks and application-specific logic. Whereas the *drivers* and *utilities* directories contained general-purpose classes for hardware abstraction. Program documentation generated using Doxygen was placed inside the *docs* directory.

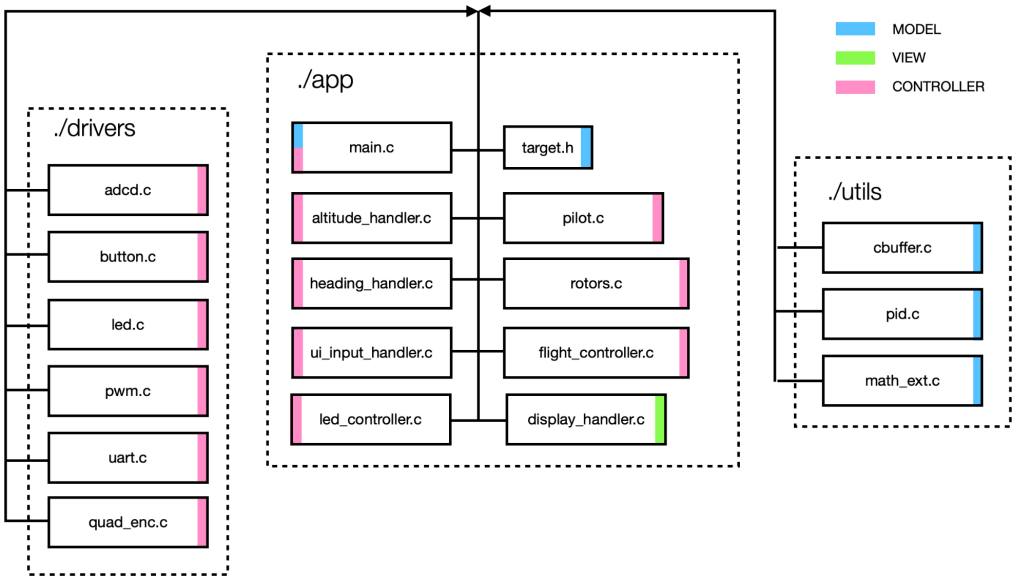

Figure 2: Block diagram of the designed program architecture.

This hierarchy was designed to maximise the code reusability across the application, and minimise module coupling. Peripheral-specific logic was contained in the respective *driver* files. For example, `button.c` managed button decoupling, long-press and double-press functionality. Driver features were further discussed in Section 2.2. Additionally, this framework helped to outline the intended purpose and interfaces between each module. This enabled parallel development pipelines to occur within the team. Whilst the *app* subdirectory contained a variety of MVC-classed components, further modularisation was considered to be unintuitive given the small scope of the application.

2.1.2 Software Architecture

The program modules outlined in Figure 2 provided a template for the functionality and scope for each module. To guide the implementation of these modules, their relationships and interfaces were formalised in a Unified Modelling Language (UML) diagram. The public methods and key member variables of each module may be seen in Figure 3.

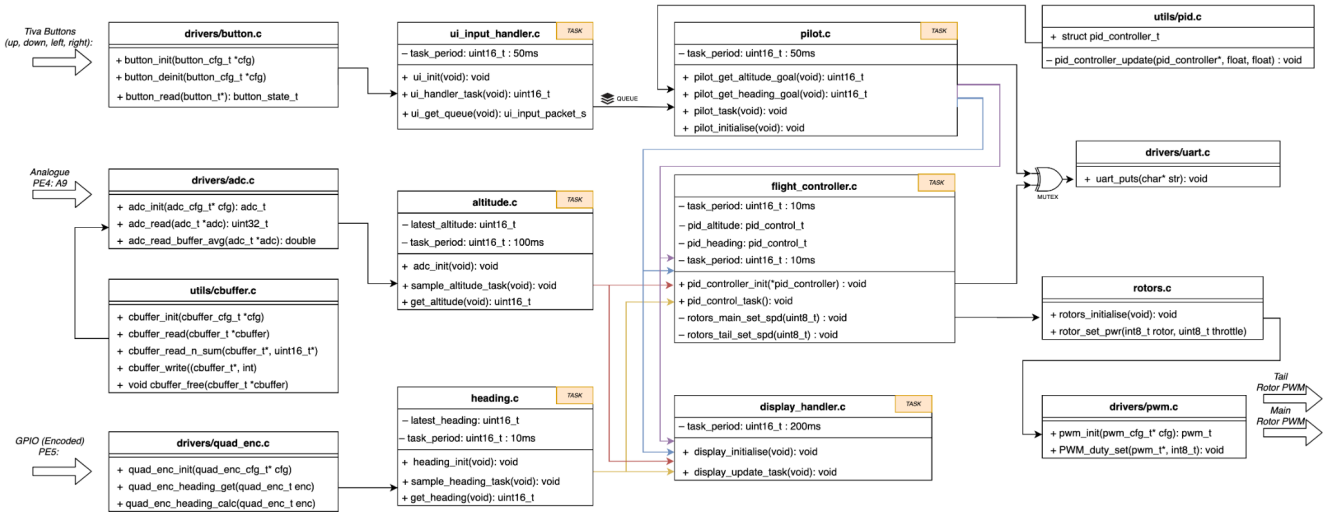

Figure 3: UML diagram of the designed program architecture.

The program architecture in Figure 3 revealed the need for accurate timing, preemption capabilities and context switching across multiple concurrent tasks. To facilitate these functionalities, and meet the specifications outlined in Section 1.2, the open-source FreeRTOS kernel was implemented as a real-time operating system. Modules with the *task* tag would be realised as FreeRTOS synchronous task threads. As synchronous tasks, each subroutine would be run every `task_period` milliseconds. A comprehensive list of each task may be found in Appendix B. This list details the parameters of each task including the description, stack size and priority.

Whilst all program tasks operated concurrently, the operation of the program can be reduced to a linear sequence of operations. The flowchart in Figure 4 illustrates this simplified operation of the program. As expected, the `ui_input_handler`, `altitude_handler` and `heading_handler` tasks were responsible for listening to the associated input signals. These tasks would then call their respective peripheral driver and then process any new samples that were available.

The *pilot* task converted user inputs such as button presses into expected heading and altitude goals. These goals were passed to the flight controller alongside the most recent heading and altitude readings. The flight controller task would then determine the error between the expected and actual readings, and update the helicopter rotor outputs to minimise these discrepancies. Finally, this information would be reflected on the OLED display.

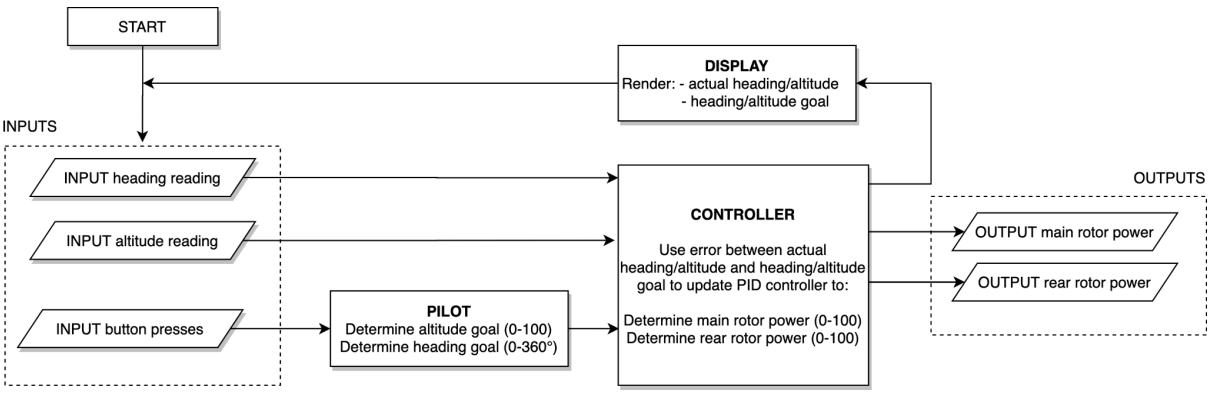

Figure 4: Flowchart of a single program loop.

These tasks were elected to run concurrently because of the varying needs and priority of each module. For example, the input modules required much higher call frequencies to sample and filter incoming data. This architecture also allowed for non-essential tasks such as the `display_handler` to have lower prioritisation.

2.1.3 Thread-Safe Operation

With six concurrent tasks operating during the program runtime, preemption-safe inter-thread communication would be essential for reliable program operation. This requirement was met through considerations to inter-thread communication, shared resources and critical sections. Firstly, the inter-thread communication interfaces were simplified through the use of adaptor patterns. Given that the controller only required the latest altitude and heading readings, these values were made available through *getter* functions. Where possible, this data was transferred as 32-bit length (or less) values. As the Tivaware TM4C123GXL operated on a 32-bit processor, all of these read/write operations were atomic and thus could not be interrupted.

Figure 5 depicts the runtime inter-thread communication of the program. As user inputs must be processed chronologically, these were pushed to a queue that was made accessible by the `pilot_task`. A FreeRTOS queue was used as each read/write operation was handled as a blocking operation. This ensured that the queue could not be preempted despite `ui_input_packet_t` being greater than 32-bits.

Several shared resources were present on the Tiva board. Most notably was the UART port. As multiple tasks may attempt to write variable-length strings simultaneously, the UART port was mutex protected. The use of a mutex prevented message corruption through mutual exclusion of the UART outgoing transmission buffer.

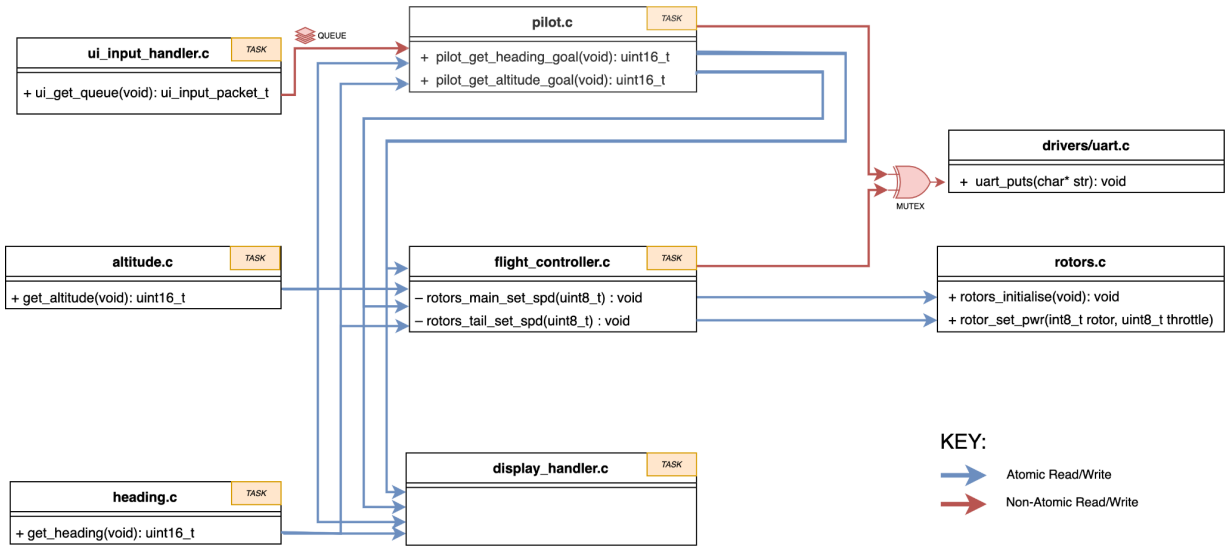

Figure 5: Block diagram of the runtime inter-thread communication between program tasks.

Finally, any read or write operations to shared variables that were non-atomic were flagged as critical sections. During the runtime of these sections, context-switching was inhibited until the critical operations were executed. As any memory allocation and communication during initialisation would occur before the kernel runtime, these were not considered to be at risk of preemption.

2.2 Peripheral Abstraction Layer

As this program was intended to run on a Tiva TM4C123GXL Development Board, low-level modules were required for the operation of the program on Tiva hardware. However, it was recognised that any explicit dependency of these low-level modules would impact the portability and flexibility of the program. To overcome this, all Tiva-specific peripheral interfaces were handled in *driver* modules. This ensured that high-level *app* modules would not depend on any low-level modules. Instead, high-level program tasks interacted only with abstractions provided by these interfaces.

2.2.1 ADC Sampling for altitude and potentiometer

As the HeliRig altitude was streamed as an analogue reading, an analog to digital converter (ADC) was required to transpose this to a digital reading. The `drivers/adcd.c` was developed to handle this peripheral interface. Once the PE4 altitude pin was initialised and enabled, the `altitude_sample_task` would use this driver to sample the analogue pin every 200ms. As the ADC conversion takes several milliseconds, the task would simply trigger a conversion to occur as a non-blocking action.

An interrupt service routine (ISR) was established to trigger on completion of the ADC conversion. Within the ISR, the ADC value would be copied from the sequencer's buffer to the circular buffer located in the SRAM. These raw values would then be filtered and used to calculate the current altitude of the HeliSim as a percentage from 0-100% within the altitude sample task.

2.2.2 Heading Sampling

The yaw of the HeliRig was measured using a rotary encoder. Similarly, the HeliSim simulator emulated a typical rotary encoder's signals by generating two out of phase signals on PB0 (Channel A) and PB1 (Channel B) of the Tiva board. In order to decode these yaw signals, a quadrature decoder was developed that encapsulated the quadrature decoding logic. This driver provided a simplified interface with an initialiser and a getter function that returned the decoded current position.

This driver was similarly implemented using an object oriented design (OOD) approach. An opaque pointer defined in the header file is used as a handle. This handle is created by the user and is used to interface with the encoder library. This not only prevents the user from modifying private attributes, but also enables extension of the library implementation without having to change any application-specific code.

The HeliRig uses an incremental rotary encoder as opposed to an absolute encoder. Due to this, only the incremental data and the relative position of the encoder could be obtained. The direction of rotation was determined by checking which channel was leading. Number of state changes can be used to determine the relative displacement of the helicopter. The absolute position of the helicopter was measured by keeping track of the relative position. The absolute yaw of the helicopter and the estimated yaw from the incremental rotary encoder can drift if state changes are missed by the microcontroller. To mitigate this issue, a reference signal is provided by most rotary encoders that can be used to calibrate out the drift. Usually, this reference signal is used as the 0° origin.

An exclusive or (XOR) between the previous and the current state of the encoder was used to detect both the direction and displacement. However, this XOR operation assumes that no state change has occurred between readings. As such, the decoding is performed using an ISR to minimise the risk of state changes going unhandled. ISRs were set up that triggered on the rising edge and falling edges of each channel. A ISR was additionally set up to trigger on the falling edge of the encoder reference signal to reset the heading back to 0° when the helicopter passes the origin.

The heading_sample_task FreeRTOS task was developed to initialise and periodically read from an instance of the quadrature decoder. Once readings were obtained, the task would make the latest helicopter yaw readings publicly-available to other program tasks through an atomic getter-function.

2.2.3 Buttons and switches Handling

It was recognised that when the buttons and switches' state change, the physical electrical connection would bounce for a finite amount of time before the connection stabilises. Using a polling scheme, the buttons and switches were sampled at 100Hz. Successive polls of an opposite state are required to be registered as a state change. Functionality such as this debouncing was encapsulated in the `drivers/button.c` and `drivers/switch.c` drivers.

This abstraction of the button and switch handling allowed tasks such as the `input_handler_task` to obtain a pointer to any required button/switches as a `switch_t` ,as shown in appendix C. This struct contains critical data about the corresponding button/switch, such as debouncing ticks and peripheral information. Therefore, any external access of data in these structures would be undesirable. To protect these structs from external access, they would be initialised in heap memory, as shown in appendix C. Subsequently, the external code can only interact with the structs with the driver functions, which protects the data being changed accidentally.

The driver only notifies the external program when a state change is detected, otherwise a `NO_CHANGE` code is returned. If a state change is detected, it would be automatically reset to `NO_CHANGE` after a call of the `button_read()` method. This ensures the external program would never miss a state change signaled by the driver. Through pre-defined polling thresholds, the button driver additionally possessed short-press and long-press differentiation.

2.2.2 Serial Communication

The gathering of runtime information such as program status logs, controller response characteristics, and CPU load was considered to be essential for the development process. To manage this functionality, a serial communication driver was developed that initialised and managed the Universal Asynchronous Receiver/Transmitter (UART) 0 port on the Tiva board. This driver allowed any task or subroutine within the program to write variable-length messages that may then be captured through a serial client on a host computer. Due to the concurrent nature of the program, multiple tasks may attempt to write to the port simultaneously. To prevent this shared resource from being accessed by multiple tasks, the *putc()* API function was mutex protected to ensure single-writer access.

2.2.2 PWM Communication

The pulse width modulation (PWM) driver module was developed to encapsulate the generation of PWM signals on the TIVA board. In the application, this driver was used by the `control_task` to control the rotors of the helicopter. Two PWM driver instances were initialised to provide individual control of the main rotor and tail rotor of the HeliRig. As the duty cycle of each PWM signal was directly proportional to the rotors' output power, these values would vary the altitude and heading of the helicopter. A PWM timer prescaler of 8 was used to set the operational frequency of 200 Hz for these PWM signals. Whilst the TM4C123GXL hardware constrained the PWM duty cycles between 2-98%, this was not found to affect the controller performance. The PWM timer used a prescaler of 8 to achieve a desired frequency of 200 Hz. Additionally, the PWM generator used an up/down counter, dividing the frequency by 2 which works as an additional clock divider.

2.3 HeliRig Control

2.3.1 Pilot

The input handling task packs all the user input elements' state into a custom struct, `ui_input_packet_t`, and sends it over to the pilot task via an RTOS queue, as detailed in appendix C. The user input task utilises the bandwidth of the queue by only sending a packet when a state change from the user input element is detected. As the sole reader of this queue, the pilot_task would translate each user input to action, such as changing altitude and heading angle. The pilot interprets a short button press as an increment or decrement of one unit. To allow for more rapid maneuverability, long presses would accelerate the unit incrementation until the user releases the button. For example, if the up button is pressed, the altitude would increase by one, and if it is held, the altitude would increase at an accelerating rate of approximately 10 units/s^2 until the up button is released.

2.3.2 PID controller

A FreeRTOS task with high priority and high update rate is created to control the helicopter. As with the majority of the program, the helicopter controller is separated to its own module for high cohesion and low coupling. Block diagram of two parallel PIDs one for heading and one for altitude is shown in Figure 6. Only two parallel PI controllers, instead of PID, are used for the project for the reasons outlined below.

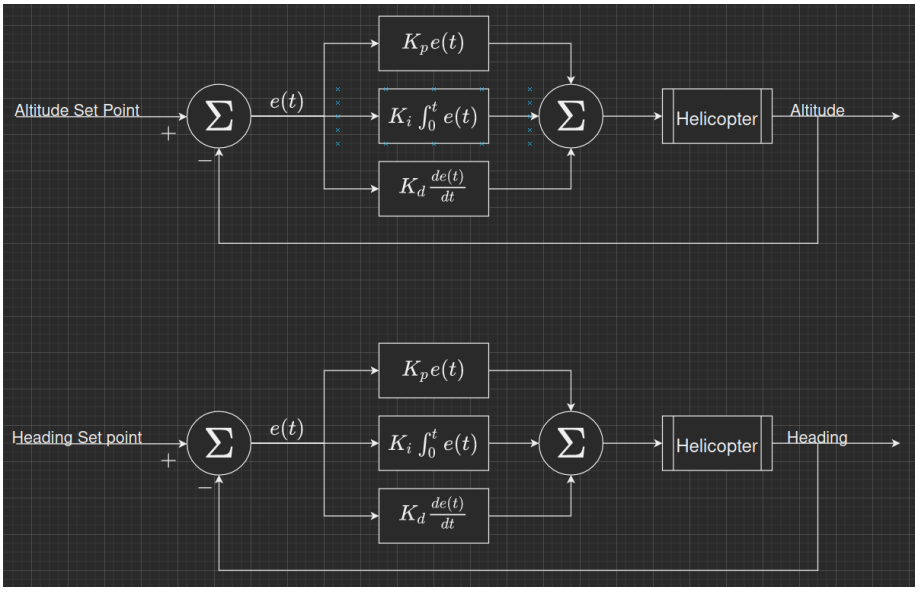

Figure 6: Block diagram of parallel PIDs used in the project

A general purpose PID controller is written similarly to the quadrature encoder and the PWM driver. The implementation of the PID is hidden from the user by using an opaque pointer. A handle is created from this opaque pointer for each PID controller. The usage of the library including creation of the PID with specific configuration, any modifications and library interfacing is done exclusively through the library with the handle. This was crucial specifically for the PID as there are a lot of private attributes such as the current integral summation that should not be modified by the user. This also allows for changing the implementation of the PID without modifying any user's code. An PID object can be created by passing in a configuration struct with fields such as proportional, integral, derivative gain.

To make the system more robust practical considerations were considered when designing the PID controller. The main and tail rotor is controlled from the PID controller through PWM by varying the duty cycle. If the PID controller's output is not limited in software, the output signal can be enormous such as 500%. In reality this is not possible and can cause undefined behaviour from the MCU. To overcome output saturation, output limiting from 2% to 98% duty cycle was implemented.

Another issue that can occur with PID controllers is what is known as integral wind up. Integral wind up is when the integral term accumulates over a slow changing system such as when the helicopter is rising to the set point. This large integral term causes the system to be slow to react to any changes when it is at the set point as other terms need to cancel out the integral term. Integral windup can also cause overshoot. To overcome this a simple anti wind up method is implemented, where the integral term is clamped to a max value in both positive and negative directions.

A PID controller module with proportional, integral and derivative gain is implemented, however only a PI controller is used for the helicopter. This was mainly due to the reliability constraints of the helicopter as derivatives can sometimes cause systems to be unstable if the gains are not properly tuned, a controller is only as good as the gains. While a PID controller would perform better in most cases compared to a PI controller, it is more time consuming and more difficult to tune the gains. Which results in increased development time and increased testing time. In most cases this can outweigh the benefits. Real life systems also exhibit noise from sensors to actuators. Differentiating these noises can lead to enormous

numbers leading to an unstable system. To overcome this, a low pass filter is implemented in software to remove high frequency noise. Usually the derivative is taken of the error, however this can lead to what is known as derivative kick. The controller can cause a sudden kick when the set point is adjusted, as this creates a derivative of the error to be instantaneously large. Derivative of the measurement instead of the error term is taken to remove the derivative kick. Derivatives can also make the system more aggressive, leading to sudden acceleration and deceleration. This is not only uncomfortable for the passengers but can also cause wear and tear of the motors, leading to a shorter lifetime. Due to all of these reasons the implemented PID controller was simplified to a PI controller for both the altitude and heading control.

The flight controller module creates and initialises two PI controller objects for heading and altitude with the desired configurations such as gains, and limits. It performs atomic reads to get current heading and altitude from the respective modules. Altitude and heading set point is read from the pilot module with atomic reads. The PI controller is updated every 10ms with the set points and the current measurements for the altitude and heading. The output signal of the PI controller is calculated through the PID library by passing in the respective handler. A modulus error between the setpoint and the current reading is used for the heading calculation. Which allows the helicopter to turn in the shortest angular displacement direction to correct for any error. The calculated PI output values are used to control the rotor with PWM through the rotors module. The entire flight controller task is set up as a critical section to avoid any erroneous calculations and controls.

The gains and parameters for the PIDs were determined experimentally using an iterative method. Initially a range of values for the proportional gain is chosen in an exponential manner from 1 to 1000. A range from 1 to 10 is chosen after observing the transient and the steady state behaviour. Linear bisection method is then used to choose an optimal value between 1 and 10. A small integral gain is then chosen iteratively to remove any steady state errors. The parameters for the PID controllers for the altitude and the heading is shown in Table 1.

Table 1: PID controller parameters for altitude and heading

Parameter	Altitude	Heading
Proportional Gain (K_p)	5	3
Integral Gain (K_i)	1	2
Derivative Gain (K_d)	0	0
Integration Summation Limit (Anti-windup)	±50	±50
Output Limit (Output saturation elimination)	2 - 98	2 - 98

2.4 User Experience

2.4.1 Input Handling

Users are able to interact with the program using four buttons, two switches and the potentiometer on the Tiva board. The user input task utilizes the button and switch peripheral abstraction drivers to poll the peripherals at 100Hz, and perform reliable debouncing and identify button presses and switch state changes. The functionality of all button and switch inputs were visualised in Figure 7.1.

The button driver's capability to identify long-presses was used by the `pilot_task` to increase the increments that button-presses yielded on the desired altitude or heading. This feature greatly improved the user experience as the altitude and heading goals could be rapidly changed using long-presses. As short-presses would still increment goals by a single unit, the rapid-changeability of goals did not come at the expense of the precision at which goals could be set.

2.4.2 User Interface (UI)

The four-row OLED display on the Orbit Booster Board was used to indicate the state of the program to the user. As seen in Figure 7.1, this was achieved by providing real-time updates of the expected (EXP) and actual (ACT) helicopter altitude and yaw. Furthermore, Figure 7.2 shows the on-board LEDs indicating high-discrepancies between the expected and actual yaw. This indication was used to acknowledge discrepancies and signify that the helicopter was in transience to rectify these errors.

A simple wave animation was run on the bottom row of the display. On top of providing a highly retro ASCII-based aesthetic, this feature was an intuitive indication of higher-than-expected CPU load. Because the display task held the lowest task priority in the program, any latency or studdering of the animation provided this insight. To improve runtime efficiency, all string-based text printed on the OLED screen was pre-defined in preprocessor macros. The display could be updated up to rates of frequency of 42 frames-per-second. To overcome the rolling-flicker caused at such a high framerate, the default automatic screen redraw settings were disabled so that the display may be explicitly re-rendered monolithically. Snippets of these methods may be found in Appendix C.

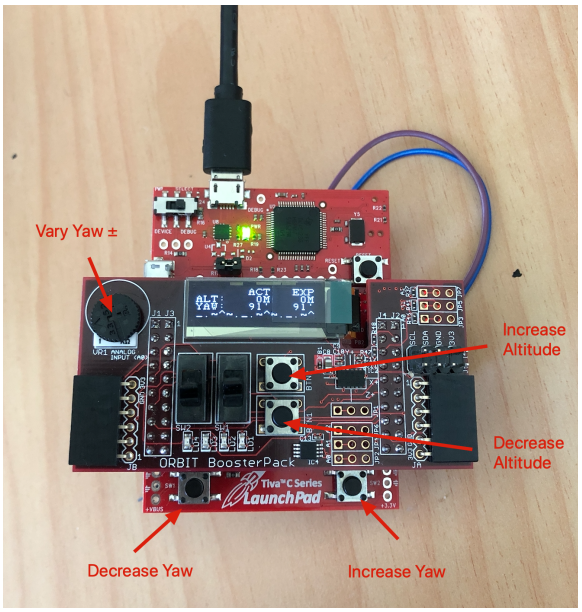

Figure 7.1: Tiva OLED Display in initial state with button functions labelled

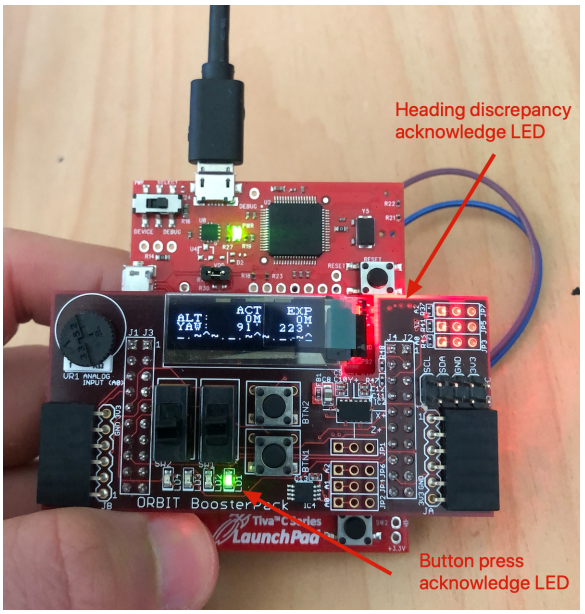

Figure 7.2: Tiva OLED display during runtime with an acknowledged high heading error.

3. Product Results and Discussion

3.1 Equivalence Partition Class - Display Module

The function of the display module is to display the input onto the OLED display. A high risk for display modules is failing to accurately display OLED, which can corrupt the overall functionality of the heli rig controller. The black-box testing technique is used to find these performance problems to improve software quality and mitigate the risk of software failures. Therefore, an equivalence partition class is applied to the display module. The chosen inputs for equivalence class were endeavoured to accomplish three types of tests: valid, borderline and invalid. Appendix 5.1 shows the lines of code in the display module that performs the preceding equivalence Class Partitioning. The test is done by manually changing the functions' parameters and documenting OLED screen output results.

As mentioned previously in Section 2.1.3, Tivaware TM4C123GXL operates on a 32-bit processor and data is transferred as 32-bit length or less values. Altitude and heading values do not drop below zero, and floating-point values present unnecessary information and take up additional text spaces. Due to these reasons, the display module receives unsigned integer type inputs to display altitude and heading.

To test that the display module functions with valid inputs, 8-bit, 16-bit and 32-bit length integer values were selected as inputs for the black box testing. Note that the input values are randomly chosen, but most importantly, are within the data length range. As for borderline testing, the inputs are the minimum and maximum values of 8-bit, 16-bit, and 32-bit length unsigned integer type values. For example, 256 is the maximum number that an 8-bit length integer can represent. Lastly, invalid testing used a variety of inputs to break the display module. The input variables can be processed as integer type variables, to allow for the display module to receive them. The inputs include floating-point, alphabetical letters and symbols.

The inputs of the display module were unsigned 32-bit integer type variables to display the altitude and heading values. In this case, black-box testing aims to compare the actual display output against the expected display output. Results and expectation explanations are tabulated in Table 2 shown below. Observe that uintX_t signalises that a variable is an unsigned X-bit length integer type.

Table 2: Equivalence Class Partitioning on Display Module.

	Input	Expected	Actual	Pass-Fail	Explanation
Valid Testing					
unsigned 8-bit integer type	(uint8_t) 1, -1	1, 255 (Overflow)	1, 255 (Overflow)	Pass	> 8-bit integers are compatible with 32-bit integers. > Negative value results in wraparound for unsigned types.
unsigned 16-bit integer type	(uint16_t) 1000, -1000	1000, 64536 (Overflow)	1000, 64536 (Overflow)	Pass	> 16-bit integers are compatible with 32-bit integers. > Negative value results in wraparound for unsigned types.
unsigned 32-bit integer type	(uint32_t) 1000, -1000	1000, 4294966296 (Overflow)	1000, -1000	Fail	> Negative value **should** result in a wraparound for unsigned types.
Borderline Testing					
Marginal Underflow	0	0	0	Pass	> Integer type can display zeros
Marginal Overflow	(uint8_t) 256 (uint16_t) 65536 (uint32_t) 4294967296	0 0 0	0 0 0	Pass	> Maximum number of values will wraparound back to zeros
Invalid Testing					
Floating Point	3.9, -3.9	3, -3	3, -3	Pass	> Floating point values as integers are rounded down
Integer as char	“1”	49	49	Pass	> Integer as character uses decimals as shown in ASCII table
Alphabetical Letter	(uint_32) A, a	65, 97	65, 97	Pass	> Character to integer conversion uses decimals as shown in ASCII table
Symbol	?	63	63	Pass	> Symbols uses decimals as shown in ASCII table
Calculation	1+1	2	2	Pass	> Equations work normally as a replacement for values

The results showed that negative unsigned 32-bit integer values are the only input parameter with an unexpected outcome. Black box testing has shown that there is a vulnerability in the module that causes the unsigned integer to be signed instead. Throughout black box testing, none of the tests noticeably broke the display module program. However, handling values with more than 3 text spaces disrupts overall text placement. Therefore, inconveniences such as 16 and 32 bits overflows must be avoided to display information clearly.

3.2 Equivalent Partition Class - Altitude and Heading Modules

Altitude and heading ADC programs output unsigned 32-bit type variables that should be locked at 0-100 and 0-360 value ranges, respectively. However, both altitude and heading programs don’t feature protection against invalid values. Therefore, the ADC programs require revisions on the display and user control modules to find potential issues between the interfaces. Note that the inputs for this equivalent

partition class are the same as the inputs in Section 3.1. This is because those inputs encapsulate valid, borderline and invalid testing.

The integration test for altitude and heading comprises two major inspections: observing the OLED display to detect any display faults and checking that buttons, switches and quadrature encoder can increment and decrement. Also, the controlled altitude and heading values should not exceed or fall under the limits. Table 3 shows the result for the equivalence partition class on ADC programs and inspecting the display and control modules.

Table 3: Equivalence Class Partitioning on interfaces between ADC, Display and User Control Modules.

		Altitude		Heading	
	Input	Display	Controls	Display	Controls
Valid Testing					
unsigned 8-bit integer type	(uint8_t) 50	Pass	Pass	Pass	Pass
unsigned 16-bit integer type	(uint16_t) 50	Pass	Pass	Pass	Pass
unsigned 32-bit integer type	(uint32_t) 50	Pass	Pass	Pass	Pass
Borderline Testing					
Marginal Underflow	0	Pass	Pass	Pass	Pass
Marginal Overflow	100 for Altitude 360 for Heading	Pass	Pass	Pass	Pass
Invalid Testing					
Floating Point	3.9, -3.9	Pass	Pass	Pass	Pass
Integer as char	“1”	Pass	Pass	Pass	Pass
Alphabetical Letter	(uint_32) A, a	Pass	Pass	Pass	Pass
Symbol	?	Pass	Pass	Pass	Pass

Moreover, a true integration test would include observing how the entire program performs in the emulator. This report excludes emulator testing due to the current lockdown situation restricting access to the heli rig emulator and lab equipment. Appendix 5.2 shows the lines of code that needs to be changed in the ADC modules to perform the preceding integration testing. Similarly to Section 3.1, the test is done by manually changing the functions’ parameters and documenting OLED screen and control output results.

3.3 Design Principles

3.3.1 Single Responsibility Principle

As outlined in Figure 1, all modules follow the single responsibility principle. Each code module held individual, and well-defined responsibilities. Additionally, relevant modules such as *drivers* were located in the same subdirectories and interfacing functions possessed consistent naming conventions. For example, all initialisation functions followed the naming convention: `module_name_initialise()`. This enabled a high level of cohesion and consistency across the program codebase. The separation of

repeatable logic blocks such as circular buffers and control elements into *utility* modules further improved the flexibility and reusability of the codebase.

3.3.2 Open Closed Principle

By closely following the single responsibility principle, the codebase was inherently compliant with the Open Closed Principle. The separation of peripheral interfacing functions into *driver* modules mitigated any hardware implementation dependence on the *task* modules. Whilst explicit polymorphism was not possible in a C-based program, extensibility was achieved through abstractions. For example, by abstracting the PID control module, a second instance could be created that extended the controllers capabilities to include heading control.

A high-level of interoperability was achieved by routing all atomic inter-module communication through getter and setter-based class functions. By following the uniform access principle, this allowed for member variables to remain private, and for modules to be more interchangeable during the development process. Through consistent interfaces, and the use of FreeRTOS, tasks could be easily interchanged to isolate and test specific modules during development.

3.3.3 Interface Segregation Principle

It was recognised that as the capabilities of the developed program increased, so too would the complexity of the program. To prevent the need to modify and append existing interfaces, individual interfaces were retained. For example, heading and altitude readings were not bundled into a monolithic function call. Instead, individual getter functions were used for each of these input datasets. This simplified the individual access of these values for the display and flight control tasks.

The compliance with this principle was best illustrated in the creation of the `pilot.c` module. As this module simply converts button presses into heading/altitude goals, it was initially considered to be redundant. However, the *pilot.c* module abstracts the `get_altitude_goal()` and `get_heading_goal()` interfaces from the input layer of the application. This simple separation enables the easy extension for user inputs from other handlers such as switches or the on-board potentiometer.

3.3.4 : Dependency Inversion Principle

The use of the hardware drivers outlined in Section 2.2 allowed for the removal of all TivaWare library includes across the *app* modules. By abstracting low-level dependencies from the high level application module, the flexibility and portability of the application was greatly improved. Furthermore, the use of an inverted dependency design pattern allowed for isolated testing of high-level modules. For this, unit test subroutines could be used to pass test data into high-level modules to perform. This enabled parallel development pipelines with lower-level modules.

3.4 Product Development Process

Program development process began by identifying the key inputs, processes and outputs of the program. Once identified, the different functions and tasks of the program were partitioned into single-responsibility modules as outlined in Figure 2. The interfaces, naming conventions and communication method was then formalised in the UML diagram seen in Figure 3. Using the module scopes and interfaces outlined in Figure 3, modules were delegated between team members and

independently developed in parallel pipelines. A bottom-up design approach was used where *driver* and *utility* modules were first developed. Completed modules were then integrated together and tested. This process allowed for a streamlined, and low-conflict development process.

All source code was thoroughly commented to increase readability and as a way of documentation. The annotated code not only helped the team to understand each other's code but also acted out as a way of communication. Documentation was periodically generated from the annotated code with Doxygen. Different team members were quickly able to understand the modules written by each other by reading this auto generated documentation.

4. Conclusion

The completed HeliRig control program was found to meet all initial requirements and objectives. Through the use of a FreeRTOS kernel, program tasks were run concurrently to control the HeliRig simulated helicopter with a closed-loop PID controller. Whilst the performance of the fight controller could not be formally assessed due to COVID-19, stable altitude and heading control was successfully achieved before lockdown. The program architecture adhered to object oriented design principles with considerations that included reusable *driver* and *utility* modules. Explicit module scopes, interfaces and naming conventions that followed SOLID design principles were formalised in UML diagrams. Having a well-defined architecture, the program was equitably developed and tested through four parallel pipelines.

5. Appendix A: Black-Box Tests

5.1 Orbit OLED Equivalent Partition Class Render Task

```
// Arrays of Valid, Borderline and Invalid Test Inputs
// Can be used to call individual input
uint32_t valid_test = [(uint8_t) 1, (uint8_t) -1, (uint16_t) 1000, (uint16_t) -1000,
(uint32_t) 1000, (uint32_t) -1000]
uint32_t borderline_test = [0, (uint8_t) 256, (uint16_t) 65536, (uint32_t) 4294967296]
uint32_t invalid_test = [3.9, -3.9, "1", (uint32_t) "A", (uint32_t) "a", "?", 1+1]

// Functions to display renderings on the OLED screen
// Each function occupies a row on the OLED screen
// Each function can call up to 2 test inputs while maintaining text placement
// Tester can update the indices of the test arrays to test different inputs
display_update_comparison_row("VAL:", "M", valid_test[0], valid_test[1], 1);
display_update_comparison_row("BDL:", "M", borderline_test[0], borderline_test[1], 2);
display_update_comparison_row("INV:", "M", invalid_test[0], invalid_test[1], 3);
```

5.2 Altitude & Heading Equivalent Partition Class Task

```
// Arrays of Valid, Borderline and Invalid Test Inputs
// Can be used to call individual input
uint32_t valid_test = [(uint8_t) 1, (uint8_t) -1, (uint16_t) 1000, (uint16_t) -1000,
(uint32_t) 1000, (uint32_t) -1000]
uint32_t borderline_test = [0, (uint8_t) 256, (uint16_t) 65536, (uint32_t) 4294967296]
uint32_t invalid_test = [3.9, -3.9, "1", (uint32_t) "A", (uint32_t) "a", "?", 1+1]

// A Getter Function to read the latest altitude that should be process as a 0 to 100
value
// returns uint32_t altitude value
// Manually change the array and its indices to test different test inputs
uint32_t altitude_get(void) {
   return valid_test[0];
}

// A Getter Function to read the latest heading that should be process as a 0 to 360
value
// returns uint32_t heading value
// Manually change the array and its indices to test different test inputs
uint32_t heading_get(void) {
 return invalid_test[0];
}
```

6. Appendix B: Task Outlines

Table B.1: FreeRTOS Task Outlines

Task handle	**Stack size**	**Priority**	**Period (ms)**	**Description**
altitude_sample_task	128	2	100	Task to initiate new analogue conversion every ALTITUDE_TASK_PERIOD ms. If a landed voltage has not been set, this task will automatically set the landed voltage to the first (smoothed) reading obtained from the altitude analogue pin.
heading_sample_task	128	2	50	Task to calculate the current heading in degrees from pulses recorded from interrupt service routine. This is also responsible for buffering and storing data so other modules can access the data.
ui_task	128	2	10	Monitor the physical hardware buttons, switches and the potentiometer. The state of these hardware elements would be sent to a RTOS queue by this task.
pilot_task	128	2	10	Read from the RTOS queue for user input states and translate the user input into actions to interact with the overall system.
flight_controller_task	256	3	10	This task is responsible for handling the PID controllers for altitude and heading. It first gets the goal and the current value for altitude and the heading. It then calculates a suitable rotor control value by updating the PIDs. The calculated value is then written to the main and tail motors through the rotors module.
update_display_task	128	1	200	Task to update the OLED display. This task gets the latest altitude & heading readings and goals. Each row is then updated and the specified changes are monolithically redrawn to prevent the rolling-shudder effect at higher refresh rates

7. Appendix C - Code Snippets

7.1 Object Oriented Design of the Button and Switch driver

In `switch.h` file:

```
//configuration struct for initialising a button struct.
typedef struct switch_cfg_s
 {
       uint32_t periph;
       uint32_t port;
       uint8_t pin;
       uint32_t activate_ticks;
} switch_cfg_t;

//button struct prototype. Implement in the .c file. content hidden from external files.
struct switch_s;
typedef struct switch_s switch_t;

/**
* @brief initialise switch by a switch_cfg_t. Returns a pointer to switch_t
*
* @param switch_cfg
* @return switch_t*
*/
switch_t *switch_init(switch_cfg_t *cfg);
```

In `switch.c` file:

```
/**
* @brief initialise switch Implementation.
*
* @param cfg
* @return switch_t*
*/
switch_t *switch_init(switch_cfg_t *cfg)
{
   //init hardware.
   switch_init_hw(cfg);

   // malloc a sw struct in heap.
   switch_t *sw_pt = pvPortMalloc(sizeof(switch_t));

   //initialize a sw struct in stack.
   switch_t sw_init = {
       .port = cfg->port,
       .pin = cfg->pin,
       .activate_ticks = cfg->activate_ticks};

   //copy the content in stack to heap.
   memcpy(sw_pt, &sw_init, sizeof(switch_t));

   return sw_pt;
}
```

7.2 UI packet implementation

In `ui_input_handler.c` file:

```
// struct sent by the UI module via queue.
typedef struct ui_input_packet_s {
 button_state_t button_up;
 button_state_t button_down;
 button_state_t button_left;
 button_state_t button_right;
 switch_state_t sw_right;
 switch_state_t sw_left;
 int32_t potentiometer;
} ui_input_packet_t;
```

7.3 displayController.c Manual OLED Redraw Methods

```
//  Initializes OLED display to only redraw when redrawDisplay() is explicitly called
void initDisplay (void) {
    OLEDInitialise (); // intialise Orbit OLED display
    OrbitOledSetCharUpdate(0); // Disable automatic redraw
}

//  Call this to manually re-draw the screen
//  This can be used to update the display without
//  any rolling flicker at up to a 50 Hz refresh rate
void redrawDisplay() {
    OrbitOledUpdate(); // Manually redraw display
}
```