ENCE361 Helicopter Project Report

Friday PM Group 2

Lei Li 49955811

Junwei Liang 91925811

Michael Redepenning 92226222

31/05/2021

1. Introduction

The aims of this project were to use a variety of embedded systems principles such as interrupts, polling, scheduling, PI control, and serial communication to continuously monitor and control a helicopter from a Tiva board. The control tasks include the helicopter's altitude, yaw angle, reliable response for button push, and stable performance in the procedure of taking off and landing.

2. Task analysis

The tasks required of the program were:

1. Read analogue voltage from the altitude sensor to continuously monitor the helicopter's altitude and convert it into a percentage of altitude value to display. The ADC samples are averaged using a circular buffer with a buffer size of 10 to eliminate the noise. As the vertical motion of the helicopter and rig is limited to < 4 Hz, a 100 Hz sample rate is sufficient for this task to respond to changes quickly.

2. Read the rotation quadrature input with reference slot and convert to degrees of yaw from reference. As the 112 slots in each disc produce 448 signals per revolution counting rising and falling edges, 900 Hz polling would be required to avoid missing steps. Since yaw change is not a regular event, 900 Hz polling would waste a lot of clock cycles. A GPIO interrupt was used to make code more efficient and more able to handle asynchronous events. When we were designing yaw interrupt, we found out that we must detect the port instead of detecting pins. Since a single pin will cause some side effects on our code.

3. Respond to buttons and switches, with software debouncing, polling, and interrupts according to the different frequency of different button events. Polling and software debouncing were used for the directional buttons. For simplicity the same frequency as the ADC sample rate was used. A GPIO interrupt happens on rising and falling edges of SW1, as it is not a frequent event.

4. Generate PWM signals to take off and hold position. Since both the main motor and tail motor are DC motors, almost any PWM frequency over 50 Hz would work. A range of 150 to 300 Hz is specified so 250 Hz was chosen.

5. Using target and real positions, use PI control to move the helicopter to the target. The output of the PI controllers must be updated at the same frequency as the ADC sample rate. This ensures accurate calculation of error for the PI controller when the analogue voltage from the altitude sensor changes. The PI controller for the tail motor needs to be updated as well to keep the helicopter's yaw position.

6. Update display and serial data. This can happen as slowly as 4 Hz for human readability without causing issues.

7. Synchronous events handling and processing. A binary flag-based system and shared data-based communication was used to handle synchronous events through SysTick interrupt instead of introducing any delay loop inside the main loop.

3. Scheduler

There are several ways to schedule real-time tasks efficiently. Round-robin scheduling processes each task in sequence, which ensures every task happens regularly but delays all tasks if one takes longer.

Interrupt scheduling only processes tasks when they are triggered by input. This saves clock cycles and power but can lead to race conditions if an interrupt is interrupted and the data it was working changes.

The method chosen is a combination of these.

For synchronous events, the SysTick interrupt routine runs at 100 Hz, the ADC sample rate, which triggers ADC reading and sets flags for button polling, sensor reading and processing, PI controller update, and display update. The ADC sample counter is used with a divider value of 10 to slow down the update for OLED display and serial transmission for human readability. The main function then performs the task and resets the flag.

This method could also be used to slow down button and ADC processing, but processing power is sufficient to run at higher frequencies and power consumption is not an issue on the helicopter rigs.

Missing asynchronous events such as yaw encoding and switch change would cause fatal errors in the control system. Yaw encoding or a change of SW1 triggers an interrupt every time an edge is detected, and the calculation for real-time position and mode is performed by the interrupt handler. This avoids unnecessary processing when the helicopter is steady.

If the program does hang, the watchdog timer performs a soft reset before the rig management alarms would shut it down.

4. Inputs

The Tiva's built in ADC converted the altitude sensor’s voltage to a digital count. This was averaged across ten samples using a circular buffer, filtering noise while still providing responsive feedback. The 0-100% altitude, a change of 0.8V from the sensor, is seen as 800 levels in the ADC, so the starting value is subtracted, and the result divided by 8 to find percentage altitude.

Yaw calculation used a lookup table to determine the direction of rotation, then converted the edge count to degrees by scaling by 0.8 (I.e., 448 counts over 360 degrees). The calculation for edge count ran inside the interrupt routine to avoid missing data and other interrupts. The edge count can increase or decrease and can be any possible integer to make sure we get the correct raw sample data, then another variable is used to keep the converted degree value for display. The value of the converted degree is in the range of [-360, 360].

Button debouncing can be done in many ways, such as waiting a set time after detecting a press. To avoid waiting, the debouncing algorithm instead of incrementing a counter for each tick is implemented. A state variable keeps the value of the current button state, if the pin read shows a new button state for 3 consecutive times from the polling, the value of the state variable will be changed.

The UP and DOWN buttons command a 10% altitude change, corresponding to -80 and +80 ADC counts respectively. The LEFT and RIGHT buttons command a 5° yaw degree change, corresponding to roughly 6 encoder edges.

5. Data communication

Inter-task communication was handled using global variables and binary flags. This is because the interrupts are asynchronous, and an ISR function that will only be invoked by interrupts will not be explicitly called anywhere in the main function.

This would affect the normal execution of the background thread and may increase worst-case interrupt latency. The ISR functions should be implemented carefully to ensure a reliable response and stable performance, so binary flags are used to avoid long ISR functions for regular events and perform fast handling of interrupts.

A finite state machine handles take-off and landing, using SW1, landed altitude, and the reference orientation to determine the state of the helicopter. The procedure of finding reference orientation locks out button input to avoid interference with the process.

Blocking serial transmission was selected to avoid interruption of other processes.

6. PI Controller

Responding to control inputs requires a control algorithm. For a free-flying helicopter, PID control is ideal for fast, accurate response to rapid changes in position and input. In this project, PI control is sufficient for the control ranges involved. Code for this function is on the next page.

Initially, two PI controllers were implemented, both at 250 Hz. A struct is used to neatly hold the various error values. The first step of tuning started from KP =0.1 and KI = 0.05, with a smaller maximum output value to protect the helicopter. Kp was gradually decrease and Ki increased by trial and error to find reliable performance, and a small offset added to the integral gain to speed up response to small errors. When stable gains were identified (Table 1) the maximum output was raised to 90%.

The initial rotation to find the reference position was attempted with a static PWM value, but this required tuning per rig so a third PI controller with limited performance was used in this mode. The desired position is incremented to always be 30 degrees ahead of the current position. Integral gain is very low to avoid windup, and the larger overshoot from the proportional gain is quickly corrected once normal operation begins.

Parameter	**Main rotor**	**Tail rotor**	**Tail rotor in ref mode**
K_P gain	0.03	0.6	1.0
K_I gain	0.03	0.4	0.0001
Max Duty Cycle %	90	60	50
Min Duty Cycle %	10	10	35

Table 1: PI controller parameters.

```
double piUpdate(PIDController_s *controller, int32_t desired_value,
                int32_t actual_value, int32_t offset) {
    /* Error signal */
    double error = actual_value - desired_value;
    /* Proportional */
    double proportional = controller->Kp * error;
    /* Anti-wind-up dynamic integrator clamping */
    double limMinInt, limMaxInt;
    /* Compute integrator Limits */
    if (controller->limMax > proportional) {
        limMaxInt = controller->limMax - proportional;
    } else {
        limMaxInt = 0.0f;
    }
    if (controller->limMin < proportional) {
        limMinInt = controller->limMin - proportional;
    } else {
        limMinInt = 0.0f;
    }
    /* Clamp integrator */
    if (controller->integrator > limMaxInt) {
        controller->integrator = limMaxInt;
    } else if (controller->integrator < limMinInt) {
        controller->integrator = limMinInt;
    }
    /* Sum them up and compute the output and apply limits*/
    controller->out = proportional + controller->integrator;
    if (controller->out > controller->limMax) {
        controller->out = controller->limMax;
    } else if (controller->out < controller->limMin) {
        controller->out = controller->limMin;
        controller->integrator += offset;
    }
    /* Integral */
    controller->integrator += error * controller->T * controller->Ki;
    /* Apply the output */
    return controller->out;
}
```

7. Conclusions and recommendations

The program met specifications, successfully flying multiple helicopters with different operating properties. Performance was slow, especially while finding reference yaw position. Larger gains would improve this, but finding stable large gains would require mathematical principles from ENME303 or much more extensive testing.

Control inputs of 10% altitude changes and 5° heading changes were not precisely followed. Correcting this would require more analysis of sensor feedback and corresponding tweaks to control.

Overshoot happened after the helicopter found reference orientation since each helicopter has a different weight, and the overshoot could not be adjusted in reference mode because control was immediately handed over to the main control scheme and the helicopter landed. This could be avoided by a second stage reference mode which drives towards the found reference.

A kernel module would be ideal to explicitly show the management of the system resources. It would also make for more readable code and easier maintenance and profiling.