Department of Electrical and Computer Engineering

University of Canterbury

ENCE 361: Embedded Systems 1

Assignment 2

Fun with avionics

By

Rhem Munro (49835619)

Ian Glass (79889718)

Friday labs 3 - 5pm

01/06/2012

Table of contents

1. Introduction (about the helicopter and PID control)

Accurate manual or automated control has always been a major issue in modern devices. Systems rarely produce a desired response from a direct input and mostly outputs are erroneous. This led to the birth of PID control where the input is transformed in order to mitigate the effect of the 'plant' on the output. PID control involves implementing mathematics on the input such as proportional multiplication (known as proportional control), numerical differentiation (known as derivative control) and numerical integration (known as integral control). Proportional control scales the input by some error which produces a response offset on the no PID control output. Integral control reduces the difference between the desired output and the actual input; however this has some latency between reaching the desired results. If the efficacy of integral control is made too large then convergence is quick however overshoot may occur. Derivative control measures the rate at which the input converges to the desired value and produces an increased impeding effect as the difference decreases. This can be very useful at reducing the overshoot caused by a brisk integral controller. The objective of this project is to design and implement code written in C to fly and stabilize a small electronic helicopter using a discrete PID control system.

2. Program design

As with any project certain criteria must be met within certain constrains. The helicopter rig has several features which are implemented to interface the stellaris LM3S1968 with the helicopter. Two outputs are used in the form of logic on or off states to communicate the relative angle of the helicopter which is read through QE (quadrature encoding) and displayed as ± degrees. Another pin is used to output a DC analog value to be read by and ADC (analogue to digital converter) which is used to determine the height as a percentage. Two more pins are used as inputs into the helicopter for main rotor and tail rotor control in the form of PWM (pulse width modulation). The Ziegler-Nichols method was chosen to determine accurate gains of our control system. The method is implemented by setting both derivative and integral gain to zero, then increasing the proportional gain to reach the 'ultimate proportional gain'. At this gain the system becomes crucially damped and an oscillatory response can be observed. Next the integral gain is increased until a satisfactory rate of convergence from the actual value to the desired value is obtained. Finally the derivative gain is increased until the desired ratio between overshoot and convergence is observed.

2.1. Kernel

2.1.1. Background round robin task scheduler

Background tasks are executed in the form of a round robin task scheduler. A round robin scheduler is a systematic perpetual execution of a list of functions or tasks. The first step was to design and implement a button de-bounce system, where by a single push or long hold of a button will only incur one effective execution of its respective task. The buttons are polled when the SysTick timer has incremented the 'poll buttons count' to a value greater than the pre-defined button pole rate. When the buttons are polled and read to be active (or pushed) a simple boolean value is set to true and the respective button cannot trigger a task until the boolean is reset. Button Booleans are only reset once the 'poll button count' is equal to zero. The Boolean reset rate must be greater than 2ms to avoid any form of button bounce. The main loop button de-bounce can be seen in figure 1.

A simple task blocking task scheduler was also implemented. The landing and take-off tasks are controlled through task blocking (set to true and false by the select button) as well as the PWM on/off outputs to the motors. The frequencies of execution of background tasks are considered to be irrelevant as none of the tasks are time critical.

2.1.2 Foreground interrupts

All the foreground tasks are implemented through vectored interrupts. The PID calculation function is called within the SysTick interrupt which runs at a pre-defined frequency. Since the PID call has a constant time value, no time calculations were needed to properly implement PID control. The QE is implemented through two separately driven interrupts. Since the run time of each QE ISR (interrupt service routine) is short compared to a consecutive state change for either of the two QE input pins, information is not lost. Figure 4 shows the implementation of the PID calculation function called within SysTick. In order to ensure interrupt atomicity the SysTick and by extension the PID function is set at the highest interrupt priority. This idea was taken from the monotonic priority principle. Data is shared between function through the use of global variables. The frequency of execution of SysTick is assumed to be the highest of all the interrupts, the two QE interrupts having a much lower frequency of execution.

2.1.3 Experimenting with background task scheduling

Experiments were done to queue tasks for the interrupts, leaving the interrupts free to continue calling their ISR's. This worked well if both the tasks where small and even better if the interrupting happen indeterminate non repetitive situation, since all interrupts would be captured. Longer running tasks where restricted in that only a set number of them where allowed on the queue. Much manipulation could be further implemented with this idea like moving tasks in the queue based on needs/priority. In one regard just by restricting the number of one type of task allowed on the queue more than another is a low form of prioritizing in a sense. Although jumping priority tasks in queue would much more elegant.

This code (fig 5.) was not uses in the final version of project because of the speed overhead and it was unnecessary.

2.2 Quadrature encoding

In order to obtain the direction in yaw angle, the phase difference between the two QE inputs must be known and compared. If phase A leads phase B then the helicopter is rotating anti-clockwise. If phase B leads phase A then the helicopter is rotating clockwise (see fig.). For our QE we used to 'state' variables to record the current state of the input pin. Our QE ran through priority pin driven edge triggered interrupts in order to satisfy the time critical characteristics of QE through preemption on background tasks. A master phase/slave phase system was implemented to reduce the complexity of code. Phase A was chosen to be the master phase in the sense that only an interrupt from U0TX (phase A) could cause an increment of 1.61 degrees in the yaw angle variable. Phase B was solely used as a comparison variable to determine the direction yaw of change. Our implementation can be seen in figure 2.

2.3 Display

2.3.1 Yaw Display

A display of the helicopter yaw is maintained on screen through a standard call of the RIT function by passing in the 'actual yaw' value. This value is also used in calculation and comparisons throughout the code. The actual yaw angle is displayed in ±degrees.

2.3.2 Analogue altitude display

The altitude display is implemented in an identical way to yaw. The actual altitude is calculated by calling a 'get ADC' function which also averages the actual altitude over 10 iterations as shown in figure 3. The new averaged altitude is then passed into the 'actual altitude' value and used in display, calculations and comparisons throughout the code.

2.4 Control

2.4.1 PWM actuator input

PWM is a much more energy efficient method of energy transfer; therefore PWM was used to drive the motors (actuators) on the helicopter. The Stellaris LM3S1968 microcontroller supports several PWM generators. Using high resolution counters an accurate repetitive rectangular pulsed wave can be produced, with a duty cycle and frequency defined by the program.

2.4.2 Altitude Control

Controlling the altitude of the helicopter required use of a single electric motor. Direct control was achieved by driving PWM into the main motor; however PID was used as an interface between the desired altitude and the PWM required to ascertain it. The user is able to freely set the desired altitude as a percentage of the maximum attainable altitude and the PID control system written within the program will adjust its PWM output until the desired altitude matches its current altitude. The PWM output is in offset from the 'middle PWM' where the control value calculated defines the direction and magnitude of offset. The range that the PWM can take is 5% to 95% duty cycle, however the PWM output can still be set to zero. Zero duty cycle will only occur when the helicopter has landed and is told (through the select button) to turn off). A single push or hold of the up or down button increments or decrements the desired altitude by 10%. This new desired value is then used in the PID function to try and match the desired altitude with the actual altitude using PID control. The desired altitude is limited to between 0% and 100%.

2.4.3 Yaw Control

Yaw control is implemented in an almost identical manner to that of the altitude control. A single push of the left or right button causes a 15 degree increment or decrement in the desired yaw. This desired yaw value is used in the PID function to try and match the desired yaw with the actual yaw using PID control. The desired yaw is limited to between -180 degrees and 180 degrees.

2.5 Take-off and landing

Take-off and landing are two separate functions that produce the opposite effect. The function increments or decrements the size of the 'middle' duty cycles on a periodic basis using the SysTick timer. However these increments or decrements do have lower and upper limits. Two results are observed, the motors start up smoothly on button push resulting in a conserved yaw angle and the motors slow down smoothly on landing for added effect. Take-off and landing functions are unblocked upon a detected select button press. Once the Take-off function has finished incrementing the middle duty cycles the helicopter is at the user's control. If select is pushed mid flight the helicopter begins its decent to land, however if the select button is pushed once again mid flight the user regains control and the helicopter will maintain its current altitude.

3. Discussion and Conclusion

The functionality of our code satisfied all the project requirements and was able to fly and stabilize the helicopter to an acceptable level. The QE and altitude measurement were accurate and produced no errors even when tested under erratic conditions. The take-off and landing protocols worked very well and the actual touch-down velocity was minimal. Improved stability during flight could have been achieved by spending more time adjusting gains. Another noticeable problem was an instantaneous increase to 95% duty cycle on the main rotor when the helicopter was lifted from the ground and the buttons were disabled. On occasion the take-off protocol would cause the helicopter to spin to its yaw limit due to the tail rotor starting on maximum duty cycle. Although these problems were apparent throughout the project no clear solution was found.

4. Appendix

```
if (G_Poll_Buttons_Count >= BUTTON_POLL_RATE)
{
      Poll_Buttons();
}
if (G_Poll_Buttons_Count == 0)
{
      Button_Debounce();
}
```

Figure 1.

```
//PA1/U0Tx (phase A) interrupt handler
void Phase_A_Handler(void)
{
      GPIOPinIntClear(GPIO_PORTA_BASE, GPIO_PIN_1);
      if (!Is_PA1_High)
      {
            G_Phase_A_State = 0;
```

```
        }
        else
        {
                G_Phase_A_State = 1;
        }
        if (G_Phase_B_State == G_Phase_A_State)
        {
                        G_Actual_Yaw += + 1.61;
        }
        else if (G_Actual_Yaw > 0)
        {
                G_Actual_Yaw -= 1.61;
        }
}
//PF5 (phase B) interrupt handler
void Phase_B_Handler(void)
{
        GPIOPinIntClear(GPIO_PORTF_BASE, GPIO_PIN_5);
        if (!Is_PF5_High)
        {
                G_Phase_B_State = 0;
        }
        else
        {
                G_Phase_B_State = 1;
        }
}
```

Figure 2.

```
void Get_ADC(void)
{
          //Bounds uiValue
          if (uiValue < ADC_Lowest)
          {
                uiValue = ADC_Lowest;
          }
          else if (uiValue > ADC_Highest)
```

```
        {
            uiValue = ADC_Highest;
        }

            G_Altitude_Average[Avg_Count] = 100 - (uiValue -
            ADC_Lowest)*100/(ADC_Highest - ADC_Lowest);

        if (Avg_Count == 9)
        {
            for (i = 0; i <= 9; i++)
            {
                  G_Actual_Altitude += (float)G_Altitude_Average[i];
            }
            G_Actual_Altitude = G_Actual_Altitude/11;
                  Avg_Count = 0;
        }
        else
        {
            Avg_Count++;
        }
}
```

Figure 3.

```
void PID_Control(void)
{
      if (G_Actual_Altitude <= 3 && G_Block_Buttons)
      {
            Motors_Off();
            G_Main_Duty_Cycle = G_Mid_Main_Duty_Cycle;
      }
      else
      {
            G_Alt_Error = G_Desired_Altitude - G_Actual_Altitude;
            G_Alt_I_Error += ((signed long)Gain_Array[2] * G_Alt_Error);

            G_Alt_Control = Gain_Array[0] * G_Alt_Error * 1000 +
G_Alt_I_Error + (Gain_Array[1] * (float)(G_Alt_Error -
G_Alt_Prev_Error))*100;
```

```
            G_Main_Duty_Cycle = G_Mid_Main_Duty_Cycle + (45 * G_Alt_Control /
G_Alt_Max_Control);

            if (G_Main_Duty_Cycle > 95)
            {
                  G_Main_Duty_Cycle = 95;
            }
            else if (G_Main_Duty_Cycle < 5)
            {
                  G_Main_Duty_Cycle = 5;
            }
      }

      G_Alt_Prev_Error = G_Alt_Error;
      PWMPulseWidthSet(PWM_BASE, PWM_OUT_0, ulPeriod * G_Main_Duty_Cycle
/100);
```

Figure 4.

```
#include "Task.h"
volatile int num_tasks_queue = 0;
void do_Task(Task **tasklist)
{
      void *null = 0;
      //first in list, an interrupt can attach a task to this anytime
      Task *task = *tasklist;
      task->funcTask(task->parm1); //do function

      //critical...we are disposing of task, if interrupt was
      //attaching to it concurently then a task is lost
      //and a memory leak occurers
      IntMasterDisable();
            *tasklist = (*tasklist)->next;
            free(task); //delete old task
            num_tasks_queue--;
      IntMasterEnable();
}
```

```
Task * new_Task(int id, void *(*funcTask)(void *p), void *parm) {
      Task *newTask = (Task *) malloc(sizeof(Task));
      newTask->id = id;
      newTask->priority = 10;
      newTask->funcTask = funcTask;
      newTask->next = 0;
      newTask->parm1 = parm;
      return newTask;
}

Task * new_Task_copy(Task task) {
      Task *newTask = (Task *) malloc(sizeof(Task));
      *newTask = task;
      newTask->next = 0;
      return newTask;
}



//if there is already one in queue then don't add another
//
//if this is done in background treat as critical section,
//if its done in an interrupt you need not worry.

Task * push_AllowOneOf_Task(Task *add, Task *first)
{
      Task *task;
      // there is nothing to add to, so this is the first
      if(!first) {
            first = add; //first in list
      }
      else
      {
            if(first->id == first->id)
            {
                  free(add); //delete all ready doing this task type id
                  return first; //early exit
            }
            task = first; //start at first
            while(task->next) {
                  task = task->next;
                  if(task->id == add->id)
                  {
                        free(add); //delete all ready doing this task type id
                        return first; //early exit
                  }
            }

            task->next = add; //put at end of list
      }
      num_tasks_queue++;
      return first;
}

//It this is done in background treat as critical section,
//if its done in an interrupt you need not worry.
void push_Task(Task *add, Task **tasklist, int allow) { //returns the first
      Task *i, *last_task;
      int count = 0;
      if(!*tasklist) {
            *tasklist = add;  // there is nothing to add to, so this is the
first
            num_tasks_queue++;
      } else {
            i = *tasklist; //first
            do {
                  if(i->id == add->id) count++;
                  last_task = i;
            } while(i = i->next);

            if((allow == -1) || (count < allow)) //'zero' is as many as
queued
            {
                  last_task->next = add; //put at end of list
                  num_tasks_queue++;
            } else free(add);
      }


}
int get_num_Tasks() {
      return num_tasks_queue;
}
```
Figure 5.