EE 472 Lab 2 (Group)
Scheduling, Digital I/O, Analog Input, and Pulse Generation

University of Washington - Department of Electrical Engineering

Introduction:

In this lab, you will develop a simple kernel and scheduler that will handle a number of tasks. Each task is required to share data using pointers and appropriate data structures (structs, arrays, etc). You will work with the Eclipse IDE development tool to edit and build your software then download and debug your code in the Make environment.

Lab Objectives:

1. Build your background in C, pointers, function pointers, dereferencing, and structs.
2. Write to and read from digital I/O ports
3. Read in an analog sensor
4. Generate pulses with accurate timing specifications
5. Build simple tasks and a task queue.
6. Build a simple scheduler and kernel
7. Use delay/timing functions

Prerequisites:

Familiarity with C programming, Eclipse and the IDE development environment, debugging, and the Make Controller board. Use the class notes, slides, text, and supplemental materials to brush up on concepts related to this lab. Make sure to cite any materials or source code used from outside this class.

Theme for Labs 2-4 and Background:

For Labs 2-4, we will be implementing a subset of an autonomous unmanned aerial vehicle (drone). Drones heavily rely on embedded systems as they take input from a variety of navigation sensors and provide timely control. For Lab 2, you will learn about building a scheduling algorithm by using external sensors and actuators to simulate controlling a drone. In Labs 3 and 4, you will control an actual model drone (The Parrot AR Drone - http://ardrone.parrot.com/parrot-ar-drone/usa/ , see Figure 1) and program autonomous maneuvers using the real time sensor data coming from the drone itself while it is in flight. You will be responsible for eventually implementing the control tasks using a network connection between the Make platform and your drone.

Figure 1: The AR Drone platform.

In this lab, we will implement a scheduler for controlling basic pulse generation, motor control, sensor input, and displaying sensor data to the LCD display.

1. Simple Scheduler

Our very first "scheduler" will be extremely simple (for practice): just a loop which sequentially calls our tasks (remember that tasks are just short functions):

```
while(1) {
    task1();
    task2();
    ...
    sync();
}
```

sync(); will be the function which keeps your loop running at a known rate. In this lab, we will do this by just wasting time in an infinite loop:

```
#define MAXLOOP <<some number>>

int i,j,k;

for(i=0;i<MAXLOOP;i++){
    for(j=0;j<MAXLOOP;j++){
            for(k=0;k<MAXLOOP;k++){
                    }
        }
}
```

We can tune the amount of delay by varying MAXLOOP or the number of nested loops. You could also use a single loop.

2. Tasks

You will have some code before the while(1) loop which performs the necessary initializations. Then the loop starts which repeatedly executes the tasks.

Write the following tasks. Each time it is called, the task must perform a small piece of its job and quickly return so that the while loop scheduler can continue to operate.

- Pulse Generation: P1 and P2
  Write two tasks to generate pulses, P1, and P2, which should appear on two output ports. They should each be 10ms in duration. Their frequency should be the same: between 50 to 100 pulses per minute, (start with 60 per minute) and P2 must follow P1 by 100ms. The frequency must be determined by the pulse value contained in a struct of type Drone_Control_Params. The task must be called with a pointer to this struct as its parameter. Don't forget to initialize the struct before the while loop. These tasks should be called each millisecond by your scheduler (i.e., your schedule will run every millisecond). You may use this fact to infer timing by counting the number of calls. The P1 and P2 signals should drive digital outputs and connect to two LED's. Use the oscilloscope to verify your timing and for debugging.

  Rationale for pulse P1 and P2: These pulses can be considered as a way to internally implement certain low-level drone controls. For instance, you have to hit the thrust motors with sequential pulses at different intervals for it to change the motor rotation and cause it to generate torque (this is what makes the drone rotate in mid air).

3. Advanced Scheduler

Consider the scheduler you built in **1** as your prototype in your lab report. The next step is to build a slightly more advanced version of your previous scheduler. In this part, we will set up some more tasks and data structures for the drone. Your code must implement all of the items described below.

This time, instead of a loop just containing your task function calls, use a single function for calling the tasks in the while(1) loop called start(). start() should have two parameters, first a function pointer to the task to start, and second, a void pointer to any parameters. Inside, start() should simply call the function with the supplied parameters. The scheduler loop would then look like this:

```
while(1) {
    …
    start(***TASK 1***);
    start(***TASK 2***);
    start(***TASK 3***);
    ...
    timer_sync();
}
```

Replace ***TASK N*** with the appropriate pointers to the function and parameters for task N. The function timer sync() can initially be commented out for testing.

After the tasks basically work, replace the delay loop from the first part in timer sync(). In this version, timer_sync() should continuously check a timer bit until 1ms has elapsed and then reset the timer and return. This way the while(1) scheduler loop will run exactly once per millisecond. Prior to the start of the scheduler loop, your code must initialize the delay timer and configure it for the correct 1ms time interval.

After you have confirmed your timing and tasks work, we are going to use a proper timer. Write an ISR routine which will respond to an interrupt from the timer. Modify the initialization routine so that the timer will cause interrupts. Finally, modify timer_sync() again so that it waits for a flag bit to be set by an ISR that you have written driven by the 1ms timer interrupt. This can just be an endless loop which checks for the bit to be non-zero, then sets it to zero and returns. The ISR will set the bit to 1 when the interrupt occurs (see the TinyTimer.zip sample code on how to use the timer).

Extra Credit (+5pt): Implement the low-level timer specified in timer.pdf for timing/counting

After you have completed the synchronized basic scheduler, start on the advanced scheduler. However, you should start designing the data structures as soon as possible in this lab.

Create data structures for Task Control Blocks (TCBs) and the TaskQueue. The TaskQueue should be initialized to contain all tasks, in order, with their status set to WAITING. Each time it is entered, the scheduler must perform the following functions.

1. Identify which task is waiting to execute next.
2. Set the status of this task to RUNNING.
3. Set the status of the previous task to WAITING .
4. Call the function pointer of this task with its parameters from the current TCB.
5. When the current task returns, start with step 1.

You may implement the TaskQueue as an array. Likely this will be an array of pointers that point to TCBs. Your TCBs should hold pointers to your tasks and any data the tasks need.

4. New Tasks

Implement the following tasks in your advanced scheduler:

- Pulse Generation
  Retain these tasks from section 1. You won't have enough digital output pins in the end, so just use the LEDs. You will want to test the output on the scope to make sure the timing still works with the new scheduler. You should adapt this task as necessary for the new scheduler.

- Sample Sensor Data
  Sample the range finder using an analog input pin and store it in a variable.

- Sample Joystick Data
  Sample the analog signals from the joystick controller that we will be providing you. You will have to assemble the joystick yourself (http://www.sparkfun.com/tutorials/161). Note that you will be allowed to use this joystick kit for your final project for controlling the drone.

- Display Sensor Data
  Display the range finder data and the joystick data on the LCD display. For the range finder, make sure to convert the ADC value to an actual distance (maybe in centimeters). Take a look at the datasheet. Convert the joystick data to a percentage (0-100% Left or Right, etc). You can either try to fit everything on the screen or flash between the values. You should use the display function you wrote in Lab1 to help with this. Because the LCD is very slow, you will have to play around with your timings a lot to get this to work. It is up to you on how you want to format the text on the display. Make sure to explain how you made this work in the report.

- Take Off Thruster Motor Speed Control (Very Basic)
  Using a variable, count the number of times the function has been called by the scheduler. Every 1,000 counts (or whatever you find reasonable), change the variable int MotorState from ON to OFF. A digital output bit must be logic 1 when MotorState is ON and logic 0 when it is OFF. Connect this bit to the pager motor on the Accessories board. We will implement a better speed control scheme in the next lab. In the next lab, you will also interface with the speed sensor to get accurate motor speed information. You may go ahead and implement that task if you want, but it is optional for this lab.

Change the motor speed based on the joystick and range finder information. Consider the joystick to be a manual override, so if the joystick is set to >5% from center (or whatever you find reasonable), you control the motor speed using the amount the joystick is moved (any direction). If the joystick is set to <=5%, then use the altimeter or range finder. As you move your hand closer to the range finder (shorter distance) the motor should spin faster. At the furthest distance, just set a fixed speed.

- Signaling

  We will want to observe the scheduler cycle via the oscilloscope. To do this we will create a little task that makes a pulse. This task should set a digital output bit, delay for 1000 loops (without returning to the scheduler), clear the bit, and return. This task must not erase or set any of the other digital output bits. This will show how well your scheduling algorithm is running.

- Timing

  The timing function should now be accomplished by an ISR driven by interrupts from the timer (or using a low-level timer). Program the timer to generate an interrupt every 1 ms. If you use another timing interval for your scheduler, make sure to justify it in your report. The ISR should count these interrupts and place the count value in a global variable Global Time. The global time could be useful for your other tasks.

5. Implementation

Remember from class that planning and designing can greatly reduce problems down the road. Sit together as a group and write out all the structs on paper. Make some block diagrams. Design a couple of the easy tasks which you can use to test the scheduler. These tasks must be very simple and short, and you must be able to easily verify that they work.

Implement your basic scheduler and make sure that the simple tasks work. Then introduce your real tasks one by one. For the advanced scheduler, lay out the TaskQueue on paper and walk through the scheduler manually to make sure you have thought through your scheduler and tasks prior to coding. Also, figure out what data you want to put in the TCBs. See “Implementing a Scheduler” on the class website under the scheduler lecture for some tips.

Get the new scheduler working first with just one trivial task. Make sure that it continues to run properly and that the first task is working. Add a couple of the more trivial tasks. A very common trap in EE472 is to divide the job up by giving a task to each member of the team. This can work well, but frequently leads to the following problem. The team sits down with this document and divides the tasks up. They go off and work independently, planning to merge their code a couple of days before the lab is due. Each team member goes off and gets their task working fine. The trouble is, when the team finally gets together to merge them for the demo, the tasks break each other somehow and there is not enough time to debug this complex system and repair the problem. Assume that debugging the complete system will take about half the available time! Make sure to have frequent sessions were everyone comes back together to merge tasks.

You may use TinyTimer as your starting base application, your previous Lab 1, or download a fresh tiny project from the makingthings website. You will have to modify the makefile to include the appropriate libraries you will need (analogin, dipswitches, etc). Make sure to include the header files in your c application as well. Take a look at the makefile in this lab and Lab 1. Also, the makingthings website briefly discusses how to incorporate libraries in your project build: (http://www.makingthings.com/ref/firmware/html/group___libraries.html)

Very important note: The Tiny firmware build has very limited functionality, whereas heavy includes a lot of features including a built in kernel/scheduler call FreeRTOS. Since you will be building your own scheduler for this assignment, you will be working off of Tiny. However, you will need certain libraries and header files from Heavy to implement some of the tasks for this assignment. One library is analogin (analogin.c and analogin.h). One problem is that when using the built in analogin.c code, it requires FreeRTOS, which in turn has its own timer. That timer can interfere with the timer interrupt you will be using. To get around this issue, we have provided our own analogin.c and myanalogin.h files which you can use in Lab2 for doing any analog input. The files and an example on how to use it are in AnalogExample.zip. Please review the example code and makefile. If you find a way to still use the timer interrupt and the built in analogin.c, I'll give you some extra credit.

6. Testing Requirements

Each team will demonstrate their results in the lab. A sign-up sheet will be posted for each team to sign up for a demo appointment. All team members must be present at the demo. All team members must also be prepared to answer a question about any part of the project. Of course many times one team member will know more than another about a specific aspect. This is OK, but it is never acceptable to know nothing about part of the code.

The following describes the most common tests we might ask you to apply to your code during the demo. The instructors are not required to ask for all of these tests, nor are the tests limited to those described below. All aspects of proper operation of the part are subject to testing by whatever means the instructor deems necessary.

- Be prepared to explain all the defined data structures and how your scheduler works.

- Pulse Generation and Timing will be verified by oscilloscope or frequency counter measurements.
- The sensor data on the LCD should update as the range changes on the altimeter and potentiometer and is it stable as other tasks are running.
- The Motor Speed Control task will be verified. Is the timing accurate? Stable and consistent? Is it disturbed by operation of other tasks? Does the motor change speed appropriate based on the sensor data?
- Signaling will also be checked via the oscilloscope.

7. Deliverables

Write up your lab report following the guideline on the course webpage. The report is due by class time on the posted due date. You are welcome and encouraged to use any of the example code on the system either directly or as a guide. For any such code you use, you must cite the source. This is an easy step that you should get in the habit of doing. Do not forget to use proper and consistent coding style; including proper comments.

Please also include the items listed below in your project report:

1. Source code (in an appendix).
   - Make sure to use a readable coding style and comment your code.
2. The final report must include what aspects of the project each team member contributed to (in an appendix).
   - Please include in your lab report an estimate of the number of hours each team member spent working on each of the following:
     - Design
     - Coding
     - Test / Debug
     - Documentation

Turn in your report and bin file to the class turn in website. Only one team member needs to do this. The assignment is due prior to class time (12:30pm) on the posted date.

Grade breakdown:

50 points total:

Sensor input and display: 10 points
Accurate timing and motor control operations: 10 points
All parts of the scheduler implemented: 10 points
Project report: 15 points
Understandable and commented code: 5 points

Check out the makingthings website for more details on the digital and analog I/O capabilities, dip switches, and the trim pot:

http://www.makingthings.com/documentation/tutorial/application-board-overview/tutorial-all-pages

The API will also be very useful for this project.

Appendix A: Using Digital Output Lines

The Make Controller board is equipped with 8 digital output lines that can be set to output a digital high or low individually. Each line is implemented as a half H-driver. This allows us to generate various signals for debugging or for driving various devices.

We control a digital output line through the set value function that is part of Make API (check out the API for more information and also how to use the Analog pins).

```
DigitalOut_SetValue(digitalOutput_i,state);
```

> digitalOutput_i – the digital output to be controlled
> state – the state (logical 0 or logical 1) to which the output is to be set

On the software side, you need to include the following header file

```
#include "digitalout.h"
```

and the following preprocessor directive. This directive should immediately follow all of your other directives; before any code or function prototypes.

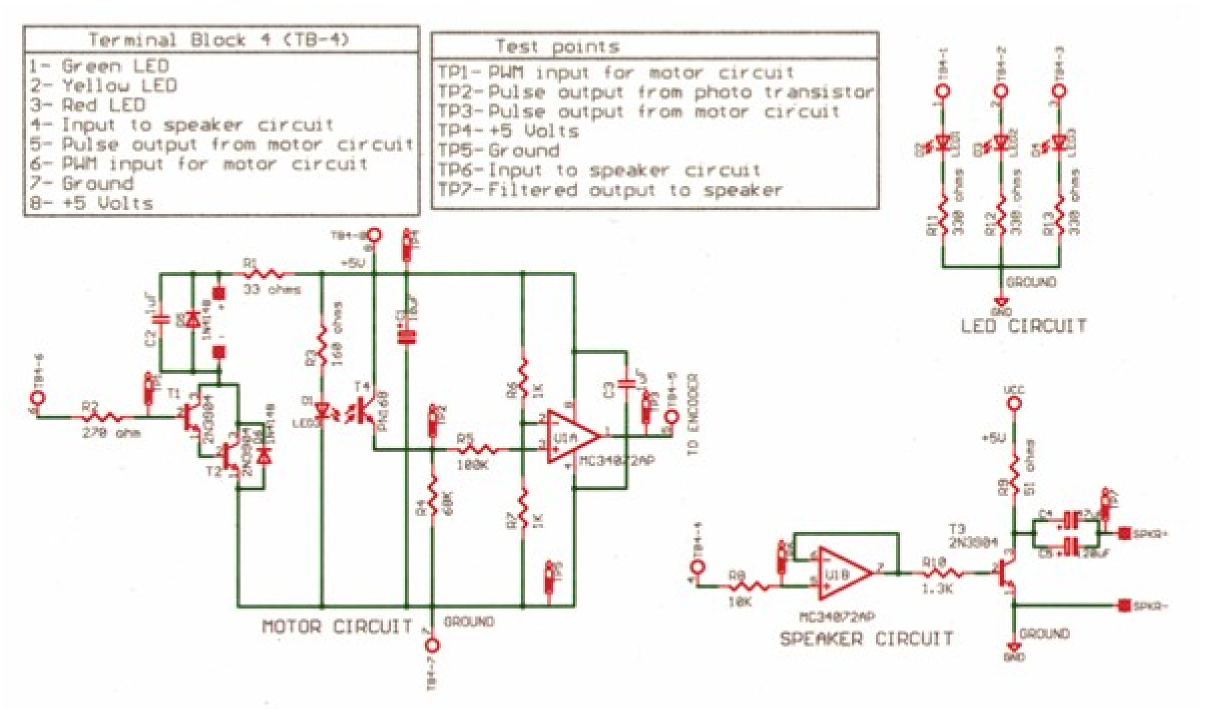
```
#undef OSC
```

Appendix B: Lab Hardware

Your lab hardware consists of the Make Controller Kit mounted on a platform that has additional external devices such as the EE472 LED and pager motor board, a keypad, and an LCD display.

- Red pushbutton for software erase..
- Four General Purpose signals which can be either inputs or outputs (Tx, Rx, CTS, RTS).
- An accessories board which contains:
  - Red, Yellow, and Green LEDs.
  - Pager motor which can be driven ON or OFF via two BJT.
  - Rotation sensor for pager motor: one pulse per revolution.
  - Audio Amplifier and Speaker.

Accessories Board
The accessories board contains several devices for our software to interact with. The schematics can be found below. Pinouts are shown for the black and white wire clamp terminal block (pin number shown in white on the PCB. The test points are small loops to which you hook your scope. The DC pager motor is wired to the two pads shown in parallel with D5. The two BJTs (T1 and T2) are biased to switch fully on or off according to an input on pin 6 of the terminal block. Note that power and ground are prewired via solder connections to the board so it is always energized when the power switch is on.



Terminal Block 4 (TB-4)
1- Green LED
2- Yellow LED
3- Red LED
4- Input to speaker circuit
5- Pulse output from motor circuit
6- PWM input for motor circuit
7- Ground
8- +5 Volts

Test points
TP1- PWM input for motor circuit
TP2- Pulse output from photo transistor
TP3- Pulse output from motor circuit
TP4- +5 Volts
TP5- Ground
TP6- Input to speaker circuit
TP7- Filtered output to speaker