CSE 190G Spring 16: Project Part 2 Detailed Instructions

Sensor interaction and real-time scheduler

Many embedded systems have been designed to cope with diverse emergency situations. For example, to safely operate vehicles, the system has to monitor information from sensors and counteract with actuators. Since it is a very time-sensitive task, the sensing management should be executed in real-time. In our project we will be designing a car monitoring and emergency detection system for which we will use the following sensors and actuators connected to RPi2:

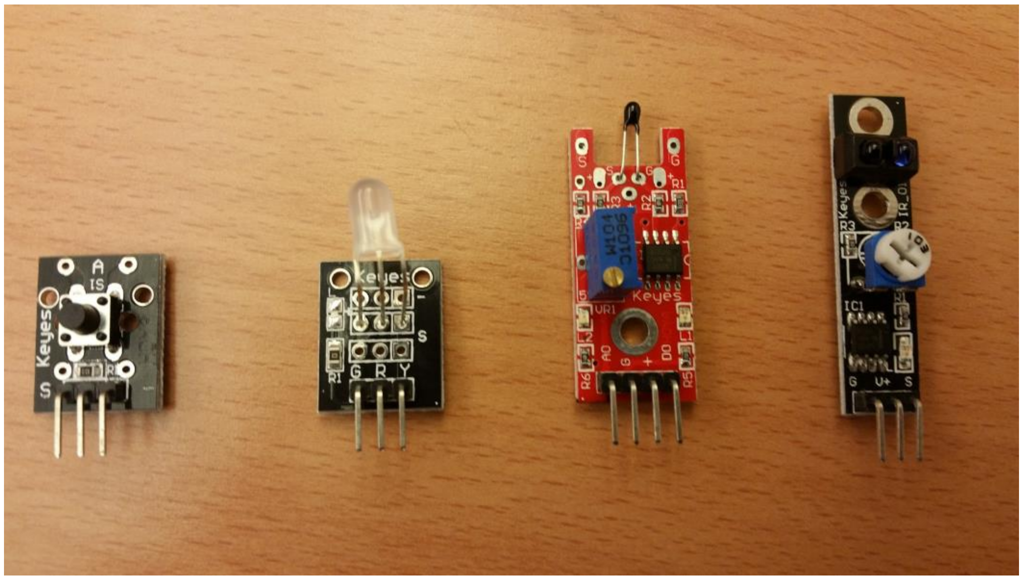
1. Button: Ignition button to start/stop the engine.
2. Two-color LED: While the engine is on, the ignition LED is also on.
3. Auto-flash LED: Warning indicator - when the engine temperature is higher than a threshold, it will start flashing.
4. RGB LED: Varies actuation when different warning/alert scenarios are seen
5. Active Buzzer: Audible alarm when temperature threshold is exceeded

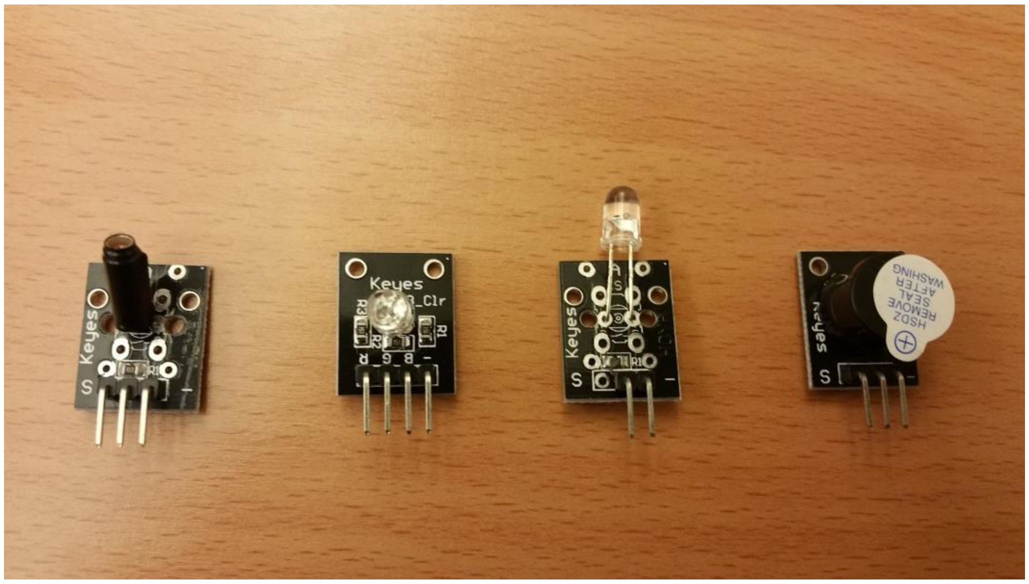
Your goal will be to connect these sensors to RPi2, develop software that correctly interacts with them and an EDF scheduler that manages the sensing program in an energy efficient manner. This instruction explains the project part 2 in two sections: (i) Sensor interaction program implementation (ii) Energy-efficient EDF scheduler implementation.

Hardware and software requirements

- Raspberry PI 2
- Intel ISA, 64 bit machine (x86_64)
- Ubuntu 64-bit 14.04.3 LTS desktop edition
    - You may run it as a virtual machine in VirtualBox.
    - For package installation commands, root access through sudo command is required.
    - If you are familiar with Linux and already have other Linux version (e.g., Ubuntu 12.04 LTS), you might get it to by using additional repos/packages, however, we cannot support that.
- Cross--compiler
- **<u>Sensor kit</u>**

  We will use 8 sensors from the sensor kit. The kit may include other sensors which have similar functionalities (e.g., Active buzzer vs. Passive buzzer), but you have to use the same sensors specified below:


(Left to right) 1. Button Switch 2. Two-color LED 3. Digital temperature 4. Tracking


(Left to right) 5. Shock 6. RGB LED 7. Auto-flash LED 8. Active buzzer

Section 1. Sensor interaction program implementation

The RPi2 provides multiple General Purpose Input/Output (GPIO) connectors for sensors. You will connect the sensors to the GPIO pins of RPi2, and implement a user-space program using WiringPi library which supports GPIO communication.

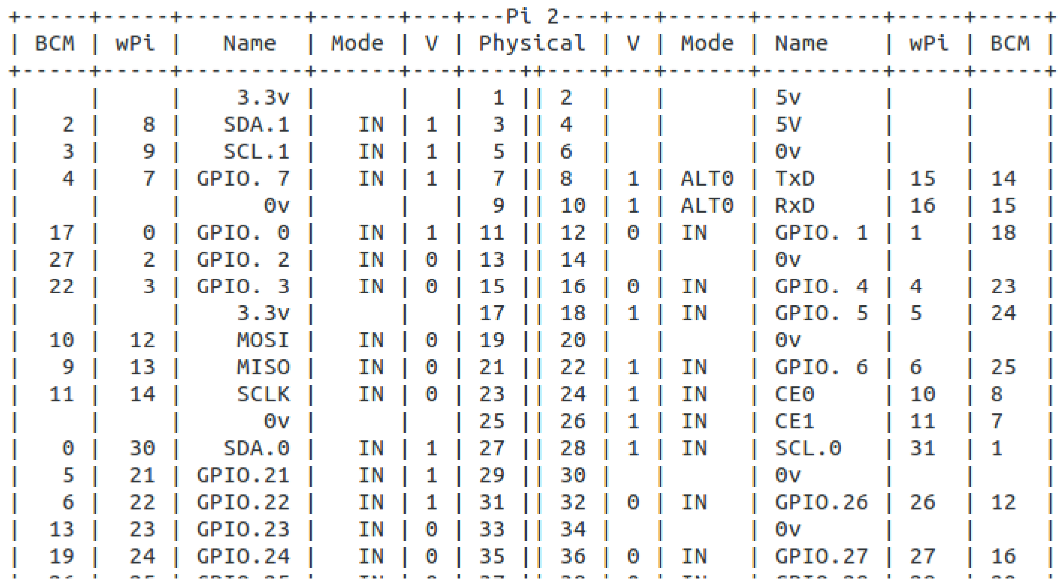
Familiarize yourself with WiringPi

In order to easily interact with sensors via GPIO, you will use WiringPI library (http://wiringpi.com/). The library provides a set of wrapper functions that control sensors connected to GPIO, so you can receive and send digital signals from/to RPi2.

First install the WiringPi library: http://wiringpi.com/download-and-install

1. Next, copy the entire folder of wiringPi to a USB pendrive.
2. Plug the USB pendrive in the RPi2
3. Mount the USB pendrive as follows (if not already mounted)
   $ sudo mkdir /media/USBDRIVE
   $ sudo mount /dev/sda1 /media/USBDRIVE
4. Copy the files of the wiringPi folder to RPi2 and build on the system:
   WIRING_PI_FOLDER$ ./build
5. To unmount the USB pendrive, type:
   $ sudo umount /media/USBDRIVE

After installing it, you can check how each GPIO pin number corresponds to the WiringPI pin number as follows:
$ gpio readall



```
+-----+-----+---------+------+---+---Pi 2---+---+------+---------+-----+-----+
| BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
+-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
|     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
|   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5V      |     |     |
|   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v      |     |     |
|   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 1 | ALT0 | TxD     | 15  | 14  |
|     |     |      0v |      |   |  9 || 10 | 1 | ALT0 | RxD     | 16  | 15  |
|  17 |   0 | GPIO. 0 |   IN | 1 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
|  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
|  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
|     |     |    3.3v |      |   | 17 || 18 | 1 | IN   | GPIO. 5 | 5   | 24  |
|  10 |  12 |    MOSI |   IN | 0 | 19 || 20 |   |      | 0v      |     |     |
|   9 |  13 |    MISO |   IN | 0 | 21 || 22 | 1 | IN   | GPIO. 6 | 6   | 25  |
|  11 |  14 |    SCLK |   IN | 0 | 23 || 24 | 1 | IN   | CE0     | 10  | 8   |
|     |     |      0v |      |   | 25 || 26 | 1 | IN   | CE1     | 11  | 7   |
|   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
|   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
|   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
|  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
|  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
```

Note that the WiringPI (wPi) numbers are different from the physical GPIO (Physical) pin numbers.
Useful information about the pins and the sample code for each sensor are available here:
http://www.sunfounder.com/index.php?c=case_incs&a=detail_&id=112&name=super%20kit

Here is how you can test the sensors:

1. Connect a sensor to proper GPIO pins
    a. A sensor has at least 2 pins. For example, an auto-flash LED sensor has two pins, one for ground (0V) and the other for input signal (0/5V). Some sensors (e.g., buzzer) may have 3 pins, one for ground (0V), another one for 5V power, and the other one for an input signal (0V or 5V). For more detailed guidelines for pins, please refer the included manual in the sensor kit.
    b. You may choose any PINs as long as the voltage requirement is satisfied. For example, you must connect 0V of GPIO to the ground pin of the auto-flash LED sensor, while connecting any GPIO PIN (e.g., 1,2,3, …) to the input of the sensor. In addition, a breadboard can help you connect the sensors more easily.
    c. Be careful when you connect the sensor. You may turn off the RPi2 to avoid any unexpected problems. Some sensors are sensitive to wrong voltage connections (e.g., 5V connection for a ground PIN), so if working connected, the sensor would be dead forever.
    d. The sensors of the Sunfounder kit already have required resistors. For example, a auto-flash led sensor already has their own resistor to properly work.
    e. The RPi2 doesn't support analog inputs/outputs on their GPIO. We do not cover analog sensing in this project. To connect analog sensors you would need an ADC (analog-to-digital convertor).

2. Implement a sensor test program in the following manner:
    a. Initialize sensor interfaces before reading/sending digital signals.
    b. Access/control a PIN's inputs and outputs.
       The following is sample code that controls an LED.

```
#include <wiringPi.h>
#include <stdio.h>

#define  LEDPin    0

int main(void)
{
  if(wiringPiSetup() == -1){//when the wiringPi initialization fails, print message to the
screen.
    printf("setup wiringPi failed !");
    return 1;
  }

  pinMode(LEDPin, OUTPUT);

  while(1){
    digitalWrite(LEDPin, LOW);   //led off
    printf("led off...\n");
    delay(500);
    digitalWrite(LEDPin, HIGH);  //led on
    printf("...led on\n");
    delay(500);
  }
  return 0;
```

3. Build and execute the sensor test program
    a. Use gcc with the WiringPi library on RPi2. E.g. on your RPi2, type this:
       `$ gcc test.c –o test –lwiringPi`
    b. To execute the program binary, you need the root privilege.
       `$ sudo ./test`
    c. If you want to cross-compile in your VM, see information below

Test each sensor prior to proceeding. Pay particular attention to the HIGH/LOW values for each sensor, as they dictate how to use it.

Optional: cross compile on Linux with WiringPi

To cross-compile programs with the wiringPi library, you also need a cross-compiled version of WiringPi library. We provide the precompiled library and header files for RPi2 in the class website.

1. Download the three files of "wiringPi_armhf" directory from the class website.
2. Place the downloaded files in the "~/RPdev/wiringPi_armhf" directory in your VM.
3. Using the downloaded files, you can cross-compile the program that uses wiringPi. For example, the above example can be cross-compiled as follows:

```
$ arm-linux-gnueabihf-gcc test.c -o test -lwiringPi -I/home/YOUR_ACCOUNT/RPdev/wiringPi_armhf/ -
L/home/YOUR_ACCOUNT/RPdev/wiringPi_armhf/
```

To test the compiled binary, copy the "test" file into your RPi2, and then execute it.

```
$ sudo ./test
```

Implement a sensor interaction program:

We will be implementing the sensing and actuation system for running and monitoring a vehicle. You will be able to turn on and off the ignition, and your code will monitor several sensors and actuate warnings/alarms for emergency scenarios as described below:

1. Before pushing the ignition button, the system is in the initial state:
    a. The two-color LED is off.
    b. The RGB LED is blue.
    c. Both other actuators (active buzzer and flashing LED) are off.
2. After pushing the ignition button, the system is in a driving state:
    a. The two-color LED is yellow, and the RGB LED is green.
    b. When in a driving state, the system handles three emergency situations as follows:
        i. If the sensor temperature is higher than a threshold of the sensor*, the buzzer plays the sound and the auto-flash LED is activated.
        When the temperature drops below the threshold, the actuators are turned off.
        ii. If the tracking sensor is activated, the RGB LED is set to the magenta color ((Red, Green, Blue) = (0x76, 0x00, 0xee)). This warning state is only resolved when the vehicle is restarted (button is toggled to off and on).
        iii. Once the shock sensor is activated, the RGB LED is set to the red color. This warning state is only resolved when the vehicle is restarted (button is toggled to off and on).
3. If ignition is pushed at any other time, the system reverts back to the initial state, i.e., (1).

*The **threshold of the digital temperature sensor** can be adjusted by the potentiometer on the sensor. Decrease the threshold by rotating the potentiometer in a clockwise direction. Check whether the digital signal is on by watching an LED on the sensor (two LEDs total will be lit when the sensor is HIGH).

Skeleton code for sensor interaction

Implement the program based on a skeleton C code that we provide. Download the files from "section1" directory of the class website. It has 6 files:

File name	***Description***
main_section1.c	The main() function of the program. Do not modify it.
assignment1.h	The source codes you need to implement for sensor interactions. You will SUBMIT them.
assignment1.c	
Makefile	A Makefile for cross compile (You need to modify a path in the Makefile)
Makefile.rp	A Makefile for native compile (i.e., on RPi2)

If you're compiling in the RPi2, move "Makefile.rp" to "Makefile"

```
$ mv Makefile.rp Makefile
```

After make it, execute as follow:

```
$ sudo ./main_section1
```

The code controls the 8 sensors using 8 different threads.

1. The main() function of "main_section1.c" executes following steps:
    a. Initializes the wiringPI and call other initialization functions
    b. Creates 8 threads corresponded to each sensor
    c. Wait until all threads are finished.
    d. Delay 1 milliseconds, and execute again the step (2) in a loop
2. During initialization, it calls the **init_shared_variable()** and **init_sensor()** functions.
   For any other data initialization, you can implement the function bodies in assignment1.c. (e.g., Initial INPUT/OUTPUT mode of the WiringPI pins)
3. Upon execution, each created thread will call the function, **body_SENSORNAME()**. You need to implement **body_SENSORNAME()** functions in assignment1.c.
    a. **Pin numbers of the sensors are already specified** in assignment1.h as macros. You have to use the specified numbers, as this is how we test your code. (e.g., #define PIN_BUTTON 0)
4. The "SharedVariable" argument is provided to the **init_shared_variable**() and the 8 **body_SENSORNAME()** functions. This variable, a C structure, is shared across all function calls and threads. You can extend this data structure as needed.

If you want to terminate the program by finishing the main loop, you can set the value of SharedVariable -> bProgramExit to 1. See assignment1.c for an existing example.

5.

The following are constraints to ensure that your implementation can be graded correctly:

DOs:

1. Complete the following functions in assignment1.h / .c:

```
void init_shared_variable(SharedVariable* sv);
void init_sensors(SharedVariable* sv);
void body_button(SharedVariable* sv); // Button sensor
void body_twocolor(SharedVariable* sv); // Two color LED sensor
void body_temp(SharedVariable* sv); // Digital Temperature sensor
void body_track(SharedVariable* sv); // Tracking sensor
void body_shock(SharedVariable* sv); // Shock sensor
void body_rgbcolor(SharedVariable* sv); // RGB LED sensor
void body_aled(SharedVariable* sv); // Auto-flash LED sensor
void body_buzzer(SharedVariable* sv); // Active buzzer sensor
```

2. Implement each body_SENSORNAME() function to control only the corresponding sensor. For example, body_button() function has to access and control only the "button sensor" with PIN_BUTTON, not the "temperature sensor".
3. Add more variables in the "SharedVariable" structure if needed.
4. Stop the program when it's executing from the terminal by pressing "Ctrl+c".

DONTs:

1. Do not submit the main file, i.e., main_section1.c. As such, this file should not be modified and your implementation must work correctly using the original file.
2. Use the predefined pin numbers for the compatibility of other RPi2 (we test your code on the TA's RPi2). Do not modify the WiringPi PIN numbers.
3. Use the original function declarations. For example, if you modify "void init_sensors(SharedVariable* sv);" to "void init_sensors(SharedVariable* sv, int SOME);", it will not be correctly executed with the provided original main file.
4. Do not implement any code that involves ANSI locking (e.g. pthread_mutex) and delay/sleep functions. This could potentially invalidate the scheduler project in Section 2.

In-Person Checkpoint Demo of Section 1 (5/16/16 during the day)

Demo your sensor interaction program to the TA. This is a sanity check for you, as your sensor interaction program will need to be correctly implemented to finish the project part 2. Bring to the demo:

- Your RPi2 with your working sensor interaction program
- Your sensors, connected to your RPi2 and ready to run with your program

A sign-up sheet will be posted for timeslots on piazza.

Build and run your sensor program with an unmodified version of main_section1.c. The TA will test the various sensing and actuation scenarios.

Section 2. Energy-efficient EDF scheduler implementation

Implement an energy-efficient EDF real-time scheduler to run sensors and actuators along with workloads that have to be executed during sensor access (e.g. sensor ramp-up, preprocessing, etc.). In Project Part 1, you already observed that the CPU frequency can be used to lower the energy cost of workload execution.

Verify your RPi2 settings

Before going into the actual implementation, verify the following settings of your RPi2, which we set in Project Part 1.

- The RPi2 must use the custom kernel from Project Part 1.
- The RPi2 must use only a single core.
- The RPi2 use the medium overclocking setting.

Change default boot option to text console

Since you will implement a real-time scheduler, your system should execute minimal additional processes. Since the window manager and graphical user interface (GUI) run a number of threads, you should limit the RPi2 to terminal mode only. If you are using GUI as the default boot option, change the boot option to the text console using raspi-config:

- $ sudo raspi-config
- Select “Boot Options”
- Select “Console Text console, requiring login (default)
- Select “Finish” and reboot your system

When executing the user-space scheduler program that you will implement, always use the text console. From the text console, you can go back to GUI:

```
$ startx
```

From the GUI, you can go back to the text console by logging out. (Menu->Shutdown->Logout)

Provided base code for EDF

The Linux system does not by default support any hard real-time scheduling mechanism. Instead, we provide the base code that can schedule threads in a user space program with frequency controls. Implement and submit the two files “assignment2.h” and “assignment2.c”.

Step 1. Download the following files from the project folder of the class website.

File name	***Description***
main_section2.c	The main() function of the program. Do not modify it.
assignment1.h/ .c	The same source code you implemented in Section 1. You will SUBMIT this.
assignment2.h/ .c	The source code you need to implement the EDF scheduler. You will SUBMIT them.
governor.h / .c	The code for CPU frequency control Do not modify it.
scheduler.h /.o	The code for scheduling threads in a user-space program Do not modify it.
workload.h	Header file for the virtual workload Do not modify it.
Makefile	A Makefile for cross compile (You need to modify a path in the Makefile)
Makefile.rp	A Makefile for native compile (on RPi2)

After downloading them, replace the assignment1.h/.c files with your implementation from Section 1.

Step 2. Download your own workload by using this link:

http://seelab.ucsd.edu/cse190g_sp16/

In that website, you will need to input your PID and download “wokload.zip”. The workload is automatically generated, so every student gets a different workload set. You will be graded based on the quality of your implementation as tested by your own unique workload and an unknown workload that we will use to compare everyone’s implementations once you submit.

Copy the zip file into the working directory that contains the files downloaded in the step 1 and unzip. For example, if you downloaded the step 1 files in “part2_scheduler” directory, place the zip file in the same directory. You can unzip the zip file as follows:

```
$ ls
assignment1.c   assignment1.h   …       workload.zip    ….
$ unzip workload.zip
```

After unzipping, there are three directories, “1”, “2”, and “3”. Each directory has has three different workloads as your test set.

```
$ ls 1 2 3
1: deadlines.c    workload.o
2: deadlines.c    workload.o
3: deadlines.c    workload.o
```

We provided the precompiled “scheduler.o” and “workload.o” files as objects instead of the source.

Step 3. Compile the provided source code

Compile the program using the provided Makefile. If you want to cross-compile, change the path in the Makefile appropriately.

```
$ make (or $ CROSS_COMPILE=arm-linux-gnueabihf- make)
```

You can see the three compiled binaries: main_section2_w1, main_section2_w2, and main_section2_w3. They were compiled using the three different workloads. If you want to compile an individual binary, you can use:

```
$ make w1
$ make w2
$ make w3
```

Let’s take a look at how the provided base code works. Using this, you can implement a soft real-time scheduler. In the same way you did in the sensor interaction program implementation, this program creates 8 threads for sensor interaction.

1. Worker thread and virtual workload

Since the actual interaction for each sensor of the sensor kit is typically executed in a very short time, we assume that there are 8 virtual workloads which precede each sensing interaction function. For example, if a sensor that needs a non-trivial access time, preprocessing, and setup/hold for ADC conversion. The provided base code mimics this concept, so the virtual workload, and subsequently, sensor access, may take hundreds of milliseconds to execute.

Once each thread which will be scheduled is executed, “workload_SENSORNAME()” function and “body_SENSORNAME()” function are executed one by one. we will call this thread the *worker thread*.

- The virtual workload is defined as “workload_SENSORNAME()” in workload.h and workload.o. The actual sensor interaction (i.e., “body_SENSORNAME()” function) will base on your section 1 implementation.
- For the actual implementation of the thread function, check out the “thread_def” macro in “main_section2.c”.

2. Scheduler thread

After initialization, the main thread of the base code plays a role for the *scheduler thread*. It calls three functions repeatedly in a loop. (See main() function in “main_section2.c” file.)

1. prepare_tasks() (closed-source in “scheduler.o”)
    a. This function is responsible for checking the period and creating the 8 worker threads.
        i. The created worker thread is not executed until it is selected by select_task() function. (see below)
    b. The deadline of each thread is same as the period.
        i. If all worker threads are executed completely, it waits until a worker thread’s execution is triggered again according to its period.
    c. For your convenience, this function also checks and reports missing deadlines (if any).
        i. It only reports first 100 errors.
2. select_task()
    a. You NEED TO IMPLEMENT this function in assignment2.c.
    b. It needs to implement a scheduling algorithm, and return a TaskSelection structure (defined in “scheduler.h”).
    c. The return variable, TaskSelection structure, has two fields:

            i. **int** task: Scheduled task (0~7: Index of the selected worker thread)
            ii. **int** freq: Applied CPU frequency (0: Low frequency, 1: High frequency)

3. execute_task() (closed-source in "scheduler.o")
    a. Based on the given TaskSelection structure, it executes the selected worker thread at the requested frequency.

The selected worker thread is executed as follows:

- The thread selected by execute_task() is guaranteed to be executed for 10 milliseconds after scheduled. Once a worker thread is executed for 10 ms, prepare_task() is executed again and another (or same) thread can be selected in select_task().
- If the selected thread finishes execution within the allotted 10 milliseconds, it terminates immediately and goes back to the loop of the scheduler thread.

You also need to consider the following facts when you're designing your scheduler:

- Even if a worker thread misses its deadline, it is not terminated immediately but still schedulable. In this situation, the scheduler considers that the deadline is missed, and doesn't create a new worker thread. You may need to schedule such threads with higher priority.
- The closed-source part of prepare_task() and execute_task() is implemented so that they run in a very short time (less than 1 ms).
- The worker threads may be not created exactly at their periods. They may have a small delay (at most 10 ms due to the allotted time slot). More specifically, the worker thread is only created when the scheduler thread is first encountered in prepare_task().

3. Program execution with adjustable run time

When executing the program, you can adjust the running time of the experiment by the program parameter.

```
$ sudo ./main_section2_w1 <time in seconds>
```

If the time is not specified, the scheduler is executed for 10 seconds.

Note again that you can test other workloads, using "./main_section2_w2" and "./main_section2_w3".

Energy-efficient EDF scheduler implementation

Implement an energy-efficient EDF scheduler in “assignment2.c” file. There are two functions you need to implement in this file:

1. void learn_workloads(struct shared_variable* sv);

This function is called when initializing the program. It will be the location where you will initialize everything you need for your scheduler. More importantly, you can characterize the workload of each worker thread here, such as the execution time of each worker thread.

Parameters

- The shared variable *sv* is the same one of Section 1. You may expand it for your purpose.

What you should know:

- You may use the “get_current_time_us();” function defined in scheduler.h. It gives the microseconds since the epoch.
- If required, you can use the PMU implementation of the project part 1 here to profile the 8 worker threads with their workloads.
- You may change the CPU frequency using set_by_min_freq() and set_by_max_freq();
    - They are defined in “governor.h/.c”.
    - To minimize the I/O time to change the frequency, the governor.c/.h uses a custom system call that is part of your custom kernel.
- The deadline (=scheduling period) of each worker thread is defined by *workloadDeadlines* in deadlines.c as an 8-element array.
    - The index of array corresponded to each worker thread is defined in “workload.h”. (i.e., the order of button, two color LED, temperature, tracking, shock, RGB LED, auto-flashing LED, and buzzer)
- If you want to estimate more accurate execution time in actual scheduling, you may need to consider the execution time of the scheduler thread as well.

2. TaskSelection select_task(struct shared_variable* sv, const int* aliveTasks, long long idleTime);

As explained earlier, this function is called while the scheduler thread is executing. You need to implement the EDF scheduler with a frequency control. Your function eventually has to return a proper worker thread to be executed with a proper frequency using the TaskSelection structure.
As a sample scheduler, we provide a naive round-robin scheduler in the provided skeleton code.

Parameter

- The shared variable *sv* is the same one of Section 1.
- *aliveTasks* is an 8-integer array which represent which worker threads are currently schedulable. If an element of the array is 1, it means the corresponded thread is still alive, so schedulable. Once a worker thread are finished, the element is set by 0.
  - For example, if BUTTON and BUZZER threads are still alive,
    alivesTasks is {1, 0, 0, 0, 0, 0, 0, 1}.
- The *idleTime* gives a time duration in microsecond. When all tasks are completely scheduled, the scheduler thread(prepare_task() function) waits for the next period without any workload. This variable gives the idle time. You will need this variable to compute the energy consumption.

What you should know:

- A rule of thumb is that you have to implement this function in a performance-efficient way. It's a part of the scheduler thread, so any delay of this function consumes the scheduling times repeatedly.
- You may use "get_scheduler_elapsed_time_us()" function to get the time since the actual scheduling. (i.e., the timer is started from the first prepare_task() call.)
- You may know which threads are newly created by using aliveTasks.
- You don't need to use the functions that controls CPU frequency inside of this function.
  - The frequency is changed by execute_task() function based on the returned TaskSelection structure. If you uses it inside of this function, it may degrade the scheduler performance.
- Don't use any file IO or interruptible function call here.
  - It may result in unexpected scheduler behavior or system blocking. Please remember again that it's a part of the scheduler which must be executed quickly.
- To avoid IO operations, you have to use "printDBG()" function instead of "printf()" function.
  - The printDBG() function is provided as a part of the base code. (see scheduler.h)
  - The usage is same to that of printf(). It stores the printed lines into a memory buffer, and the buffer is flushed when the program is finished.
  - The memory buffer size is 1 MB at default. You can adjust the buffer size in main.c: init_deferred_buffer() function.

Energy computation

Use the table below to calculate the CPU energy values:

CPU Frequency	**Power (Active)**	**Power (Idle)**
900MHz	800 mW	50 mW
600MHz	400 mW	50 mW

The active power is the power consumption while any workload is working. The idle power is the power consumption while no workload is working. The idle time is provided in the third parameter of the select_task() function.

Project Submission Instructions (by 23:59:59 PST, May 25)

Submit the following via TED:

- Submit the four files “assignment1.c, assignemnt1.h, assignment2.c, and assignemnt2.h”
    - Don’t submit other source files.
    - Your code must be executed correctly using the predefined wiringPI PIN numbers and the provided other files.
- Report
    - Maximum 3pgs, 12pt Times New Roman font, excluding figures and table.
    - Briefly explain how you implemented the sensor interaction program in Part 1
    - Carefully explain how you designed your energy efficient scheduler
    - Provide a table for the three provided workloads. The table should contain the estimated energy and the result if the scheduler was missing the deadline or not
    - Do not include your source code in the report.

Troubleshooting

If any step quits prematurely and shows an error message, log the error and read it. Some common errors can be figured out with a quick web search. You may also post on the class discussion board (anonymously if you want), or email the TA. To help us debug your problem, include the error message and output of the following commands (include packages.txt as an attachment)

- uname-a
- lsb_release-a
- echo $PATH
- dpkg --get-selections > ~/packages.txt