Lab 4 & 5

Final Project Report

Sanghoon Han

Daniel Jang

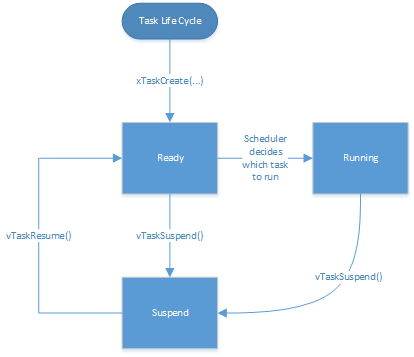
Daniel Park

**I. Introduction**

This is the final lab project of the quarter. The main goals of labs 4 and 5 are replacing the old scheduler with that of FreeRTOS, implementing multi-train intersection, adding a break temperature warning system, and implementing the web server functionality to enable communication between a web browser and the MCU. All required specifications have been met and the final project is fully functioning without error.

**II. Design Description**

**a) Porting to FreeRTOS**



The project now is implemented using the FreeRTOS scheduler. The scheduler relies on defined tasks with priorities. We have divided the project into multiple separate .c files which consists of the tasks. The tasks are vScheduleTask, vTrainComTask, vSwitchControlTask, vSerialComTask, and vCurrentTrainTask. There is a function that can be used to create a task:

Creating tasks:

xTaskCreate( TASK\_FUNCTION, TASK\_NAME, STACK\_SIZE, FUNCTION\_POINTER, TASK\_PRIORITY, TASK\_HANDLER);

The following priorities are:

TrainComTask: 3

SwitchControlTask: 2

CurrentTrainTask: 1

SerialComTask: IDLE (0)

ScheduleTask: IDLE (0)

At every clock-edge the scheduler is looking to run a non-suspended task. If there are more than one available, it runs the one with higher priority. Although this is all done in series, it is so fast that it may seem like all the tasks are ran in parallel. In our case it is important for SwitchControlTask to run before the CurrentTrainTask. This is because the SwitchControlTask uses flags and checks to see what state the trains will be in next.

Previously the timing of our system was done using the hardware timer delay. In this lab we used the FreeRTOS delay function. The ScheduleTask’s main function is to time the system and also resume tasks. By resuming tasks every 500 ms we are able to keep track of time and behavior of train. The suspension of the tasks are done at the beginning of the while loop of each task. In main.c the TaskHandles are defined and accessed through extern in the functions.

*Task Handles:*

In main.c:

xTaskHandle xTrainComHandle;

xTaskHandle xCurrentTrainHandle;

xTaskHandle xSwitchControlHandle;

xTaskHandle xSerialComHandle;

In tasks:

extern xTaskHandle xTrainComHandle;

extern xTaskHandle xCurrentTrainHandle

extern xTaskHandle xSwitchControlHandle

extern xTaskHandle xSerialComHandle

*Resuming Tasks:*

In schedule.c:

if (trainCount > 0) {

vTaskResume(xSwitchControlHandle);

vTaskResume(xCurrentTrainHandle);

}

vTaskResume(xSerialComHandle);

In interrupt IntGPIOe (push button):

xTaskResumeFromISR(xTrainComHandle);

*Suspending Tasks:*

In tasks:

while(1)

{

vTaskSuspend(TaskHandle)

...

...

}

*Semaphores*

In this lab we used a semaphore handle to ensure that the switch control precedes current train task. At the end of switch control task there is a xSemaphoreGive(xSemaphore) call. At the start of current train there is a xSemaphoreTake(xSemaphore, portMAX\_DELAY). If the pair of train com task and switch control tasks are ready to be executed, there is no way that the current train task can be executed because it has not received a semaphore from the switch control.

**b) Multi-train Intersection** (how is it implemented)

The smart train management system allows there to be up to five trains. However only one train will be moving at a time. While there can be four train only two of them will be displayed: 1) The current train that is traversing and 2) the next train waiting.

This was implemented using a queue of train. The data structure our team decided was an array of trainInfo structs. The struct will contain the following variables:

typedef struct {

bool\* toE;

bool\* toW;

bool\* toS;

bool\* toN;

bool\* fromE;

bool\* fromW;

bool\* fromS;

bool\* fromN;

int\* passengerCount;

unsigned int\* trainSize;

unsigned int\* initializedTime;

} trainInfo;

trainInfo trainQueue[5];

This trainQueue can be accessed outside the scope using the extern call similar to the TaskHandle.

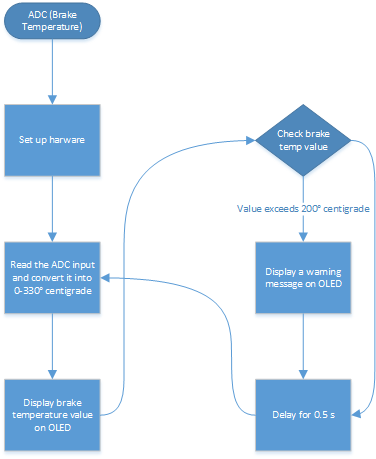
When a train is done traversing the elements in the trainQueue moves over one. The train in the 0th position of the array is replaced by the train in the 1st position, train in the 1st position gets replaced with the 2nd element train, and so forth. The 4th element train gets replaced by an “empty” train. This is a train that has all its fields initialized to 0 or FALSE.

**c) Brake Temperature Measurement & Display**

The train system monitors the brake temperature of the train currently traversing the intersection. Brake temperature data is obtained using an an analog-digital convertor (ADC). The data is then displayed on the OLED. All of this occurs in real time.

The brake temperature is simulated by measuring voltage across a potentiometer. We predefine 3.3 V across the potentiometer as 330° centigrade, and we defined 0 V as 0° centigrade. However, the ADC records data as a 10-bit number (1024 in integer form). A quick mathematical conversion allowed us to model brake temperature correctly.

The ADC can be set to one of four sequences. Different sequence samples the input at different intervals. Because the brake temperature only needed to read a constant dc voltage, the train system’s ADC was set to use sequence 3. If we wanted to read an sine wave or some other non-constant function, we would’ve had to use a different sequence to capture the data.



**d) Changes in LED / UART Display**

The LED and Hyperterminal now displays two trains: The current train and the next train waiting in the queue. This is displayed side by side vertically. This is implemented in the LED by manually printing using coordinates. However in the UART display we must print it line by line. So essentially it will be printing informations for 0, 1, or 2 trains. We simplified the print by making our own printing function that only requires the string:

// print to UART

void printUART(unsigned char \* s)

{

int size = 0;

while (\*(s + size))

{

size++;

}

UARTSend(s, size + 1);

}

For each line there will be a check if there is one or greater than one train in the queue:

if (trainCount = 0)

{

clear screen

}

else if (trainCount > 0)

{

print train2 information

if (trainCount > 1)

{

print train2 information

}

}

**e) Web Server**

The MCU is capable of functioning as a web server, serving series of hard coded hex values (in *httpd-fsdata.c*) that translate into HTML files. Because manipulating an array full of hex codes to create an HTML file is a greatly painful process, we wrote a java program that translates an HTML file into an array of hex codes as represented in the pseudocode below:

public main(args) {

File file = new File(HTML\_File\_DIR);

Scanner scanner;

try {

scanner = new Scanner(file);

List list = new ArrayList;

while (scanner has next line) {

String line = scanner.nextLine();

for(i = 0; i < line.length; i++) {

char letter = line.charAt(i);

list.add(“0x” + toHex(letter));

}

list.add(“0xd”);

list.add(“0xa”);

}

scanner.close();

list.add(“0”);

System.out.println(list);

} catch(exception) {

handle exception;

}

}

Using the above program, we were able to freely create HTML files and translate the file to a format the board requires.

To turn on the web server, an IP address has to be given to the *uipIP\_ADDR* constants in *uIP\_Task.c*. Our choice of IP address is *128.95.141.163*. The *mainINCLUDE\_WEB\_SERVER* variable in main() also has to be turned to 1 so the web server task is activated.

The function that processes input from the browser is present also in *uIP\_Task.c* as *vApplicationProcessFormInput()*. This function reads the HTTP request sent from the browser and determines what request was made. If it reads that a direction variable has been inserted in the request, then it turns on the respective direction flag and resumes the TrainCom task, which introduces new trains to the wait queue. The following lines of code perform the job:

if (strstr(pcInputString, "dir=e")) {

fromState = 8;

} else if (strstr(pcInputString, "dir=w")) {

fromState = 4;

} else if (strstr(pcInputString, "dir=s")) {

fromState = 2;

} else if (strstr(pcInputString, "dir=n")) {

fromState = 1;

}

vTaskResume(xTrainComHandle);

**III. Results**

The final version of the lab project meets all requirements and seems to contain no error. All tasks resume and suspend according to the design logic. Interrupts are handled properly and without error. Up to five trains can be stored in a queue representing the waiting line for the trains. Interaction with the web server meets all basic requirements.

**IV. Time Spent**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Sanghoon | Daniel Park | Daniel Jang |
| Design | 15 | 10 | 15 |
| Coding | 10 | 18 | 12 |
| Debugging | 15 | 16 | 15 |
| Testing | 5 | 4 | 5 |
| Documentation | 5 | 2 | 3 |
| Total | 50 | 50 | 50 |

**V. Conclusion**

In this project, the lab group migrated the existing system over to FreeRTOS, implementing task handlers and semaphores in the process. Additionally, break temperature warning system was implemented using an analog-digital converter (ADC) and the train queue was completely re-done from scratch to allow maximum of five trains waiting to cross the intersection. Last but not least, the board was turned into a basic web server to allow communication between a web browser and the board. All features were tested multiple times to ensure proper functionality. There was no error found in the course of the project.