Assignment 3: RTOS

Objective

The purpose of this assignment is to get hands-on with FreeRTOS, a famous open-source real-time operating system. The point of this assignment is to give you practical knowledge in setting up a real-time operating system and defining tasks for execution.

After completing this assignment, you will be able to:

- Set-up and compile FreeRTOS
- Create tasks in FreeRTOS
- Practically demonstrate scheduling of several tasks in FreeRTOS

Getting the Environment

- It is recommended to use the free software Visual Studio Community for this assignment. The FreeRTOS system you are about to setup will execute in Visual Studio Community, and output from the system is available from its console.
- More help about setting up both Visual Studio Community and Eclipse with the FreeRTOS project are in the accompanying document with this lab called "How To Setup the FreeRTOS Project in Visual Studio Community"

Create Repo by Accepting Assignment in this link:

https://classroom.github.com/a/E-MRRXPO

Deadline is extended to Dec 15th as discussed in Lecture

Part 1: A Hello World Example

In this part, we will make a Hello World example in FreeRTOS.

1) Go into FreeRTOS->Demo->Win32-MSVC folder. Start the Win32.vcxproj file which will execute Visual Studio. Visual Studio will import all the files that are relevant to the project. All these file can be seen to the left in the Solution Explorer.

2) To compile the project, you simply click the play button in the panel. This will compile all the files that are included in the project template, and once that is done, it will execute the project.

Q1) When you execute the project, What do you get? What is the reason for that?

3) Now, add a "Hello World" task to the project. To do so, we go to the main function in main.c and look for the task scheduler function. We don't have yet any tasks to find. So, that is what we need to do next. To do so, create a void function called Hello Task and inside this function, create an infinite loop as follows:

```
void HelloTask(){
   while (1){
        printf("Hello World!\n");
   }
}
```

4) You need to add the period to the task and that we do by adding a delay to the task. So, add the delay of one second or 1,000 milliseconds.

```
void HelloTask(){
   while (1){

        printf("Hello World!\n");
        vTaskDelay(1000);
   }
}
```

5) Now, we add the task to FreeRTOS. And that is done by first creating a task handle and then use the xTaskCreate function.

```
int main( void )
{
   /* This demo uses heap_5.c, so start by defining some heap
regions.  heap_5
   is only used for test and example reasons.  Heap_4 is more
appropriate.  See
   http://www.freertos.org/a00111.html for an explanation. */
   prvInitialiseHeap();

   /* Initialise the trace recorder.  Use of the trace
recorder is optional.
   See http://www.FreeRTOS.org/trace for more information. */
   vTraceEnable( TRC_START );

   xTaskHandle HT;
   xTaskCreate(HelloTask,"HelloTask",configMinimal_STACK_SIZE,
   NULL,1,&HT);
   vTaskStartScheduler();
   for (;;);
   return 0;
}
```

Q2) Figure out what is the meaning of the parameters passed to the xTaskCreate function

Part 2: Multiple Tasks

1) You are required to create two tasks with the following parameters (hint look at xTaskCreate() in the API):

```
Task1 name="Task1"Task2 name="Task2"
Task1 stack size = 1000
Task2 stack size = 100
Task1 priority = 3
Task2 priority = 1
```

2) Create task functions containing the following functionality

- Task1 should print out "This is task 1" every 100 milliseconds (hint use fflush(stdout) after printf())
- Task2 should print out "This is task 2" every 500 milliseconds

Q3) Provide

1) a screenshot of the execution in a report and
2) the file containing the source code of your solution called FreeRTOS-Part2.

Part 3: Task Priorities

In this part, we will **focus on task priorities.** We have one high intensity task which uses much of the CPU and one sparse intensity task which mainly uses I/O. The task here is to handle the priorities of the tasks to make sure that no tasks miss the deadline. You will also learn how to measure time in FreeRTOS to evaluate the feasibility of a real-time system in practice.

After completing this part of the assignment, you will be able to:

- Handle priorities explicitly in FreeRTOS
- Measure time in FreeRTOS
- Use call back hooks in FreeRTOS

1) Create a task "matrixtask" containing the following functionality:

```
#define SIZE 10
#define ROW SIZE
#define COL SIZE
static void matrix_task()
{
   int i;
   double **a = (double **)pvPortMalloc(ROW * sizeof(double*));
   for (i = 0; i < ROW; i++) a[i] = (double *)pvPortMalloc(COL * sizeof(double));
   double **b = (double **)pvPortMalloc(ROW * sizeof(double*));
   for (i = 0; i < ROW; i++) b[i] = (double *)pvPortMalloc(COL * sizeof(double));
   double **c = (double **)pvPortMalloc(ROW * sizeof(double*));
   for (i = 0; i < ROW; i++) c[i] = (double *)pvPortMalloc(COL * sizeof(double));

   double sum = 0.0;
   int j, k, l;

   for (i = 0; i < SIZE; i++) {
          for (j = 0; j < SIZE; j++) {
                 a[i][j] = 1.5;
                 b[i][j] = 2.6;
          }
   }

   while (1) {
          /*
          * In an embedded systems, matrix multiplication would block the CPU for a
long time
          * but since this is a PC simulator we must add one additional dummy
delay.
          */
          long simulationdelay;
          for (simulationdelay = 0; simulationdelay<1000000000; simulationdelay++)
                 ;
          for (i = 0; i < SIZE; i++) {
                 for (j = 0; j < SIZE; j++) {
                        c[i][j] = 0.0;
                 }
          }

          for (i = 0; i < SIZE; i++) {
                 for (j = 0; j < SIZE; j++) {
```

```
                        sum = 0.0;
                        for (k = 0; k < SIZE; k++) {
                               for (l = 0; l<10; l++) {
                                      sum = sum + a[i][k] * b[k][j];
                               }
                        }
                        c[i][j] = sum;
                 }
          }
          vTaskDelay(100);
    }
}
```

2) Create a task "communicationtask" containing the following functionality:

```
static void communication_task()
{
    while (1) {
           printf("Sending data...\n");
           fflush(stdout);
           vTaskDelay(100);
           printf("Data sent!\n");
           fflush(stdout);
           vTaskDelay(100);
    }
}
```

3) Create the tasks in FreeRTOS with the task creation call:

```
xTaskCreate((pdTASK_CODE)matrix_task, (signed char *)"Matrix", 1000, NULL, 3,
&matrix_handle);
xTaskCreate((pdTASK_CODE)communication_task, (signed char *)"Communication",
configMINIMAL_STACK_SIZE, NULL, 1, &communication_handle);
```

Q4) “communicationtask" must send a simulated data packet every 200ms but is often blocked by matrixtask, fix this problem without changing the functionality in the tasks.

4) Create a new task "prioritysettask" which:

- Sets the priority of "communicationtask" to 4 in case its execution time is more than 1000 milliseconds (Hint: look at vApplicationTickHook() to measure it)
- Sets the priority of "communicationtask" to 2 in case its execution time is less than 200 milliseconds (Hint: look at vApplicationTickHook() to measure it)

Q5) Provide a screenshot of the execution and answer the following questions in a report:

- Why is "matrixtask" using most of the CPU utilization?
- Why must the priority of "communicationtask" increase in order for it to work properly
- What happens to the completion time of "matrixtask" when the priority of "communicationtask" is increased?
- How many seconds is the period of "matrixtask"? (Hint: look at vApplicationTickHook() to
- measure it)

Part 4: A periodic Task

In this programming assignment, you will handle aperiodic jobs.

1) Implement the task "matrixtask" from Part 3.
2) Add a software timer in main() to trigger a software interrupt every 5 seconds. (Documentation found Here.)
3) Define a Timer callback function outside main() with the following functionality:

```
/* A variable to hold a count of the number of times the timer expires. */
long lExpireCounters = 0;
void vTimerCallback(TimerHandle_t pxTimer)
{
    printf("Timer callback!\n");
    xTaskCreate((pdTASK_CODE)aperiodic_task, (signed char *)"Aperiodic",
configMINIMAL_STACK_SIZE, NULL, 2, &aperiodic_handle);
    long lArrayIndex;
    const long xMaxExpiryCountBeforeStopping = 10;
    /* Optionally do something if the pxTimer parameter is NULL. */
    configASSERT(pxTimer);
    /* Increment the number of times that pxTimer has expired. */
    lExpireCounters += 1;
    /* If the timer has expired 10 times then stop it from running. */
    if (lExpireCounters == xMaxExpiryCountBeforeStopping) {
          /* Do not use a block time if calling a timer API function from a
          timer callback function, as doing so could cause a deadlock! */
          xTimerStop(pxTimer, 0);
    }
}
```

4) Create an aperiodic task using the following functionality:

```
static void aperiodic_task()
{
    printf("Aperiodic task started!\n");
    fflush(stdout);
    long i;
    for (i = 0; i<1000000000; i++); //Dummy workload
    printf("Aperiodic task done!\n");
    fflush(stdout);
    vTaskDelete(aperiodic_handle);
}
```

Q6. The following questions should be solved with programming and the questions should be answered in a report:

- **Is the system fast enough to handle all aperiodic tasks? Why?**
- **If not, solve this problem without alter the functionality of any task**
- **What is the response time of the aperiodic task?**
- **Provide a screenshot of the running system**