



TOULOUSE SITE

Final assignment

GOETHAL Julien

Promo: *Aéro 5*

Class: *5TS1*

Course: *Real time embedded system* – Professor: *M. SINGH*

Submission date: *January 14, 2024*

Contents

1. Abstract	1
2. Introduction	1
3. Methods	3
3.1 Task 1 : say "Working" when everything works.	3
3.2 Task 2 : Fahrenheit to Celsius converter.	5
3.3 Task 3 : Multiplying two long int.	6
3.4 Task 4 : Binary search.	7
4. Results	8

1. Abstract

Real-time System is a system which is used for performing some specific tasks. It is a computational system which is used for various hard and soft real-time tasks. These specific tasks are related with time constraints. The tasks assigned to real-time systems need to be completed in given time interval. The tasks assigned to real-time systems need to be completed in given time interval. Embedded Systems are integrated systems which are formed by the combination of computer hardware and software for a specific function. It can be said as a dedicated computer system which has been developed for some particular reason. But it is not our traditional computer system or general purpose computers, these are the Embedded systems which may work independently or attached to a larger system to work on few specific functions. These embedded systems can work without human intervention or with a little human intervention. The embedded systems which are designed to perform real-time tasks are known as Embedded Real-time Systems or Real-time Embedded Systems. So in a few words, a real-time embedded system is a specialized computing system designed to perform specific tasks in a timely manner. It operates in an environment where the correctness of its operations depends not only on the logical result but also on the time at which the results are produced.

2. Introduction

In this final assignment, the main objective is to design an RTOS with the following tasks :

- Periodic task 1 : print "Working" or some other string which says everything is working as normal.
- Periodic task 2 : Convert a fixed Fahrenheit temperature value to degree Celsius.
- Periodic task 3 : Define any two long int big number and multiply them, print the result.
- Periodic task 4 : Binary search a list of 50 elements (fix the list and element to search.)

As my computer is working on Windows, I made this assignment using Oracle VM to code in C. Before going into detail for the 4 tasks I have to made, let's firstly explain how can we perform these tasks. We got a file called *ipsa_sched.c* which allows us to reproduce what does the main program inside the **Posix_GCC** folder but with the 4 different tasks.

On this picture on the next page, I've made several changes, first inside the red rectangle, I define a priority for each of the four tasks I have. These number are totally random meaning that we didn't necessarily need to set the priorities like that. But still, we can understand something interesting in it. Here the task 4 has the highest priority meaning that it will be the only receiving task working. Obviously the send task priority will also execute itself, without that we cannot see if our code is working because every receiving task has to wait the "signal" from the send task to perform itself.

```

95
96 /* Priorities at which the tasks are created. */
97 #define mainQUEUE_RECEIVE_TASK_PRIORITY1 ( tskIDLE_PRIORITY + 2 )
98 #define mainQUEUE_RECEIVE_TASK_PRIORITY2 ( tskIDLE_PRIORITY + 3 )
99 #define mainQUEUE_RECEIVE_TASK_PRIORITY3 ( tskIDLE_PRIORITY + 4 )
100 #define mainQUEUE_RECEIVE_TASK_PRIORITY4 ( tskIDLE_PRIORITY + 5 )
101 #define mainQUEUE_SEND_TASK_PRIORITY ( tskIDLE_PRIORITY + 1 )
102
103 /* The rate at which data is sent to the queue. The times are converted from
104 * milliseconds to ticks using the pdMS_TO_TICKS() macro. */
105 #define mainTASK_SEND_FREQUENCY_MS pdMS_TO_TICKS( 500UL )
106 #define mainTIMER_SEND_FREQUENCY_MS pdMS_TO_TICKS( 2000UL )
107
108 /* The number of items the queue can hold at once. */
109 #define mainQUEUE_LENGTH ( 2 )
110
111 /* The values sent to the queue receive task from the queue send task and the
112 * queue send software timer respectively. */
113 #define mainVALUE_SENT_FROM_TASK ( 100UL )
114 #define mainVALUE_SENT_FROM_TIMER ( 200UL )
115
116 /*-----*/
117
118 /*
119 * The tasks as described in the comments at the top of this file.
120 */
121
122 static void prvQueueReceiveTask( void * pvParameters );
123 static void prvQueueReceiveTask2( void * pvParameters );
124 static void prvQueueReceiveTask3( void * pvParameters );
125 static void prvQueueReceiveTask4( void * pvParameters );
126 static void prvQueueSendTask( void * pvParameters );

```

Figure 1: Call and priority assignment to tasks.

Inside the green rectangle I create the 4 received task in order to be able to test them. Again we highlight the fact that if we don't change the task priority, only the last received task will be perform, so to see the others as well, we can just change the number inside the definition of priority (The higher the number will be, the more priority the task will be).

```

/* Start the two tasks as described in the comments at the top of this
 * file. */
xTaskCreate( prvQueueReceiveTask, /* The function that implements the task. */
"Rx", /* The text name assigned to the task - for debug only as it is not used by the kernel. */
configMINIMAL_STACK_SIZE, /* The size of the stack to allocate to the task. */
NULL, /* The parameter passed to the task - not used in this simple case. */
mainQUEUE_RECEIVE_TASK_PRIORITY1, /* The priority assigned to the task. */
NULL ); /* The task handle is not required, so NULL is passed. */

xTaskCreate( prvQueueReceiveTask2, /* The function that implements the task. */
"Rx", /* The text name assigned to the task - for debug only as it is not used by the kernel. */
configMINIMAL_STACK_SIZE, /* The size of the stack to allocate to the task. */
NULL, /* The parameter passed to the task - not used in this simple case. */
mainQUEUE_RECEIVE_TASK_PRIORITY2, /* The priority assigned to the task. */
NULL ); /* The task handle is not required, so NULL is passed. */

xTaskCreate( prvQueueReceiveTask3, /* The function that implements the task. */
"Rx", /* The text name assigned to the task - for debug only as it is not used by the kernel. */
configMINIMAL_STACK_SIZE, /* The size of the stack to allocate to the task. */
NULL, /* The parameter passed to the task - not used in this simple case. */
mainQUEUE_RECEIVE_TASK_PRIORITY3, /* The priority assigned to the task. */
NULL ); /* The task handle is not required, so NULL is passed. */

xTaskCreate( prvQueueReceiveTask4, /* The function that implements the task. */
"Rx", /* The text name assigned to the task - for debug only as it is not used by the kernel. */
configMINIMAL_STACK_SIZE, /* The size of the stack to allocate to the task. */
NULL, /* The parameter passed to the task - not used in this simple case. */
mainQUEUE_RECEIVE_TASK_PRIORITY4, /* The priority assigned to the task. */
NULL ); /* The task handle is not required, so NULL is passed. */

xTaskCreate( prvQueueSendTask, "TX", configMINIMAL_STACK_SIZE, NULL, mainQUEUE_SEND_TASK_PRIORITY, NULL );

```

Figure 2: Creation of the different tasks with parameters.

Here I create the 4 received tasks and the sending task with their own parameters (their name and their priority associated).

This ends the global changes of the *ipsa_sched.c* code, let's now come into detail and see precisely what was implemented for each of the four tasks.

3. Methods

In this section, we will show and explain how the four different tasks are made.

3.1 Task 1 : say "Working" when everything works.

This first task is telling us to print 'Working' or a similar word meaning that our code is correctly working. So the main thing to implement here is to be able to print 'working'.

```
static void prvQueueReceiveTask( void * pvParameters )
{
    uint32_t ulReceivedValue;

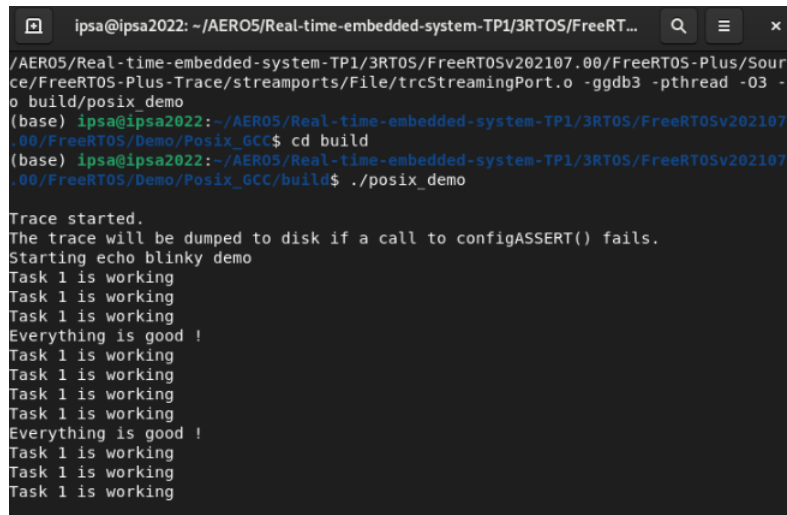
    /* Prevent the compiler warning about the unused parameter. */
    ( void ) pvParameters;

    for( ; ; )
    {
        /* Wait until something arrives in the queue - this task will block
         * indefinitely provided INCLUDE_vTaskSuspend is set to 1 in
         * FreeRTOSConfig.h. It will not use any CPU time while it is in the
         * Blocked state. */
        xQueueReceive( xQueue, &ulReceivedValue, portMAX_DELAY );

        /* To get here something must have been received from the queue, but
         * is it an expected value? Normally calling printf() from a task is not
         * a good idea. Here there is lots of stack space and only one task is
         * using console IO so it is ok. However, note the comments at the top of
         * this file about the risks of making Linux system calls (such as
         * console output) from a FreeRTOS task. */
        if( ulReceivedValue == mainVALUE_SENT_FROM_TASK )
        {
            console_print( "Task 1 is working\n" );
        }
        else if( ulReceivedValue == mainVALUE_SENT_FROM_TIMER )
        {
            console_print( "Everything is good !\n" );
        }
        else
        {
            console_print( "Unexpected message\n" );
        }
    }
}
```

Figure 3: Code for task 1

Here the main difficulty is to understand how to print what we want to. The command `console_print` is a function that basically print (it did something similar to `std::cout`) so we just have to print "Task 1 is working". Everything here is related to the rate of data arrival : I set the `mainVALUE_SENT_FROM_TASK` to 0.5s meaning that every 0.5s, my console will print "Task 1 is working" (if everything works naturally). I also set the `mainVALUE_SENT_FROM_TIMER` to 2s meaning that every 2s, my console will print "Everything is working". Finally, if none of these conditions statements are met, my console will produce an error message.



```
ipsa@ipsa2022: ~/AEROS/Real-time-embedded-system-TP1/3RTOS/FreeRT...
/AEROS/Real-time-embedded-system-TP1/3RTOS/FreeRTOSv202107.00/FreeRTOS-Plus/Sour
ce/FreeRTOS-Plus-Trace/streamports/File/trcStreamingPort.o -ggdb3 -pthread -O3 -
o build/posix_demo
(base) ipsa@ipsa2022:~/AEROS/Real-time-embedded-system-TP1/3RTOS/FreeRTOSv202107
.00/FreeRTOS/Demo/Posix_GCC$ cd build
(base) ipsa@ipsa2022:~/AEROS/Real-time-embedded-system-TP1/3RTOS/FreeRTOSv202107
.00/FreeRTOS/Demo/Posix_GCC/build$ ./posix_demo

Trace started.
The trace will be dumped to disk if a call to configASSERT() fails.
Starting echo blinky demo
Task 1 is working
Task 1 is working
Task 1 is working
Everything is good !
Task 1 is working
Task 1 is working
Task 1 is working
Task 1 is working
Task 1 is working
Everything is good !
Task 1 is working
Task 1 is working
Task 1 is working
```

Figure 4: Task 1 output.

Here and even if we cannot see the refresh of the code because it's a picture, the first task seems to work perfectly (again to make it work, I fixed to the first task the highest priority).

3.2 Task 2 : Fahrenheit to Celsius converter.

The second task was about giving the temperature in Celsius from an input temperature in Fahrenheit. Here, the main objective is to apply the converting formula of Fahrenheit to Celsius which is given by the following :

$$T(C) = \frac{5}{9} \times (T(F) - 32)$$

So to test this formula, I set a random Fahrenheit temperature and then I apply the previous formula to it before printing the conversion thanks to the following code :

```
static void prvQueueReceiveTask2( void * pvParameters )
{
    uint32_t ulReceivedValue;

    /* Prevent the compiler warning about the unused parameter. */
    ( void ) pvParameters;

    for( ; ; )
    {
        /* Wait until something arrives in the queue - this task will block
        * indefinitely provided INCLUDE_vTaskSuspend is set to 1 in
        * FreeRTOSConfig.h. It will not use any CPU time while it is in the
        * Blocked state. */
        xQueueReceive( xQueue, &ulReceivedValue, portMAX_DELAY );

        /* To get here something must have been received from the queue, but
        * is it an expected value? Normally calling printf() from a task is not
        * a good idea. Here there is lots of stack space and only one task is
        * using console IO so it is ok. However, note the comments at the top of
        * this file about the risks of making Linux system calls (such as
        * console output) from a FreeRTOS task. */
        if( ulReceivedValue == mainVALUE_SENT_FROM_TASK )
        {
            console_print( "Task 2 is working\n" );
        }
        else if( ulReceivedValue == mainVALUE_SENT_FROM_TIMER )
        {
            int temps_in_fh = 32+rand() % 50;

            double temps_in_dg = (temps_in_fh -32)/9.0*5.0;
            console_print( "température en Fahreneit : %d F, conversion en degré : %2f°C\n", temps_in_fh, temps_in_dg );
        }
        else
        {
            console_print( "Unexpected message\n" );
        }
    }
}
```

Figure 5: Fahrenheit to Celsius code.

As we can see, I've put my variable *temps_in_dg* as a double to be able to get the result with accuracy. Beside this, I'm only applying the formula and printing the result :

```
Task 2 is working
Task 2 is working
température en Fahreneit : 59 F, conversion en degré : 15.000000°C
Task 2 is working
Task 2 is working
Task 2 is working
température en Fahreneit : 47 F, conversion en degré : 8.333333°C
Task 2 is working
Task 2 is working
Task 2 is working
température en Fahreneit : 75 F, conversion en degré : 23.888889°C
Task 2 is working
Task 2 is working
Task 2 is working
température en Fahreneit : 67 F, conversion en degré : 19.444444°C
Task 2 is working
Task 2 is working
Task 2 is working
```

Figure 6: Task 2 output

3.3 Task 3 : Multiplying two long int.

For this task we have to multiply two long int numbers and print them. Let's show the code for this exercise :

```
static void prvQueueReceiveTask3( void * pvParameters )
{
    uint32_t ulReceivedValue;

    /* Prevent the compiler warning about the unused parameter. */
    ( void ) pvParameters;

    for( ; ; )
    {
        /* Wait until something arrives in the queue - this task will block
         * indefinitely provided INCLUDE_vTaskSuspend is set to 1 in
         * FreeRTOSConfig.h. It will not use any CPU time while it is in the
         * Blocked state. */
        xQueueReceive( xQueue, &ulReceivedValue, portMAX_DELAY );

        /* To get here something must have been received from the queue, but
         * is it an expected value? Normally calling printf() from a task is not
         * a good idea. Here there is lots of stack space and only one task is
         * using console IO so it is ok. However, note the comments at the top of
         * this file about the risks of making Linux system calls (such as
         * console output) from a FreeRTOS task. */
        if( ulReceivedValue == mainVALUE_SENT_FROM_TASK )
        {
            console_print( "Task 3 is working\n" );
        }
        else if( ulReceivedValue == mainVALUE_SENT_FROM_TIMER )
        {
            long int a= 519195165119;
            long int b= 784816654984;
            console_print( "a*b =%ld\n",a*b );
        }
        else
        {
            console_print( "Unexpected message\n" );
        }
    }
}
```

Figure 7: Task 3 code

And this is the output for this task :

```
Task 3 is working
Task 3 is working
a*b =2882928388848657272
Task 3 is working
Task 3 is working
Task 3 is working
Task 3 is working
Task 3 is working
a*b =2882928388848657272
Task 3 is working
Task 3 is working
Task 3 is working
Task 3 is working
Task 3 is working
a*b =2882928388848657272
Task 3 is working
Task 3 is working
Task 3 is working
Task 3 is working
Task 3 is working
```

Figure 8: Task 3 output

3.4 Task 4 : Binary search.

Last, this code should help us to do binary search : to remind ourselves, binary search is a common method used to search a specific element into a sorted list of element. It operates by repeatedly dividing the search range in half until the target is found or the search space is exhausted. This logarithmic time complexity makes it significantly faster than linear search, especially for large datasets (even if in this case, we only have 50 elements in the list). Additionally, binary search requires a sorted array to function effectively. Here is my implementation of binary search :

```

if( ulReceivedValue == mainVALUE_SENT_FROM_TASK )
{
    console_print( "Task 4 is working\n" );
}
else if( ulReceivedValue == mainVALUE_SENT_FROM_TIMER )
{
    int y[50] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50};
    int search_n = 10;

    int taille = sizeof(y) / sizeof(y[0]) / 2; // taille de la liste :2
    int i = y[taille];
    int compteur = 2;

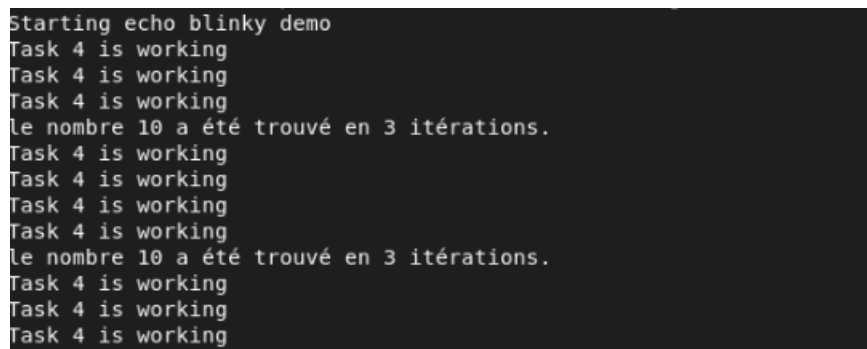
    while (i != search_n) {
        if (i < search_n) {
            taille = taille + taille / compteur;
            i = y[taille];
            compteur++;
        } else if (i > search_n) {
            taille = taille - taille / compteur;
            i = y[taille];
            compteur++;
        }
    }

    console_print( "le nombre %d a été trouvé en %d itérations.\n", i, compteur-1 );
}

```

Figure 9: Task 4 code

I set the number to find to 10, and I also determine the number of iteration needed to find 10 in my list :



```

Starting echo blinky demo
Task 4 is working
Task 4 is working
Task 4 is working
le nombre 10 a été trouvé en 3 itérations.
Task 4 is working
Task 4 is working
Task 4 is working
Task 4 is working
le nombre 10 a été trouvé en 3 itérations.
Task 4 is working
Task 4 is working
Task 4 is working

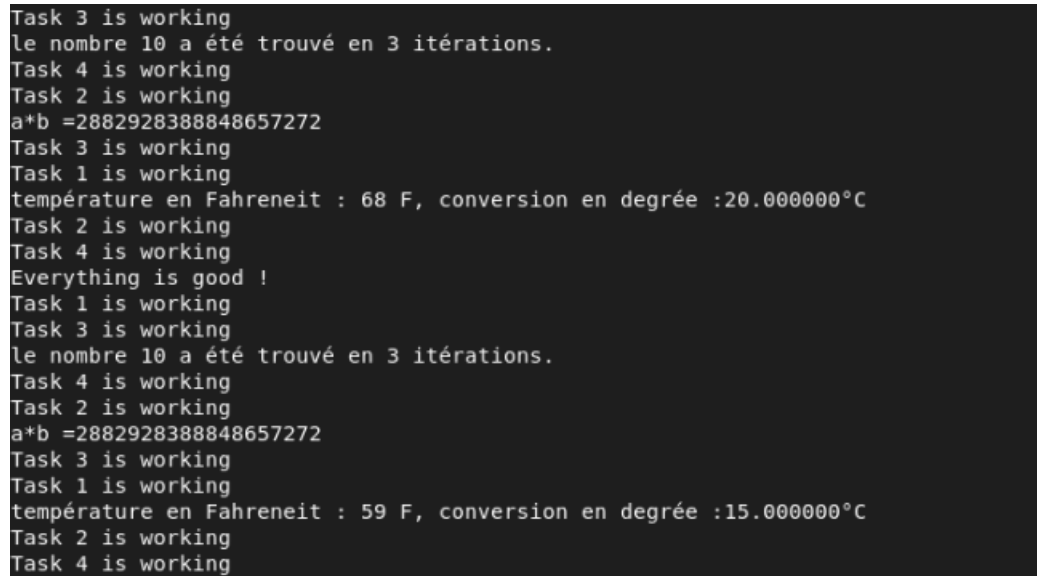
```

Figure 10: Task 4 output

So here as it is say, the number is find in only 3 iterations which is few when we look at the size of the list, so that example shows the strenght of this type of algorithm.

4. Results

Now that we have seen separately our four task, let's see how they act when we give them the same priority :



```
Task 3 is working
le nombre 10 a été trouvé en 3 itérations.
Task 4 is working
Task 2 is working
a*b =2882928388848657272
Task 3 is working
Task 1 is working
température en Fahreneit : 68 F, conversion en degré :20.000000°C
Task 2 is working
Task 4 is working
Everything is good !
Task 1 is working
Task 3 is working
le nombre 10 a été trouvé en 3 itérations.
Task 4 is working
Task 2 is working
a*b =2882928388848657272
Task 3 is working
Task 1 is working
température en Fahreneit : 59 F, conversion en degré :15.000000°C
Task 2 is working
Task 4 is working
```

Figure 11: Global Output

So here, we see that every of the four task are working and giving their results. They are following the same pattern everytime. This ends the final assignment report.