



**Master CORO M1**  
**Master Commande et Robotique**  
**Master in Control and Robotics**

**EMARO+ M1**  
**European Master on Advanced Robotics**

**Project Report**  
<24/06/2020>

**<Operating systems for Real Time Processing>**

<Varad Pradeep Diwakar>  
<Vinayak PRASANNA KUMAR>

**Supervisor(s)**

<Audrey Queudet>, <ECN>

## Contents

1.Introduction and Objective.....	4
2.FreeRTOS.....	5
2.1Introduction.....	5
2.2Setup.....	5
2.3Task Creation .....	8
2.4Starting the scheduler.....	9
2.5Creating a periodic task.....	9
2.6Semaphore creation and task synchronization .....	10
3.ChibiOS.....	13
3.1Introduction.....	13
3.2 Setup for ChibiOS/RT .....	15
3.3 Task Creation .....	16
3.4 Creating a periodic task.....	17
3.4.1 Make a Project Folder .....	17
3.4.2 Adding headers.....	17
3.4.3 Writing Make File .....	18
3.4.4 Allocating a working area to thread. ....	18
3.4.5 Declaring a thread function.....	18
3.5 Task creation and scheduling .....	18
3.5.1 Explanation of Scheduling and pre-emption of threads.....	20
3.5.2 Steps to execute the program. ....	21
3.5.3 Results.....	22
3.6 Semaphore creation and task synchronization .....	22
3.6.1 Adding header file.....	23
3.6.2 Writing a Make file. ....	23
3.6.3 Create aperiodic Task.....	23
3.6.4 Semaphore.....	24
3.6.5 Result .....	26
4. RTEMS.....	27
4.1 Introduction.....	27
4.2 Steps for installation.....	28
4.2.1 Setting up environment .....	28
4.2.2 Select installation prefix.....	28
4.2.3 Getting Source code .....	29
4.2.4 Install Tool suite.....	29

4.2.5 Bootstrap the RTEMS Sources .....	30
4.2.6 Build a Board Support Package (BSP) .....	31
4.2.7 Build Hello World application .....	34
4.3 Thread Creation: .....	39
4.4 Semaphore creation and synchronization: .....	40
5. Basic installations .....	43
6. Conclusion .....	44
7 References .....	45
8 Appendix .....	46

## 1. Introduction and Objective

The backbone of every real time system is the RTOS which is responsible for scheduling of tasks and to ensure that each deadline is met as per the requirement. There exist multiple vendors/ corporations which create and supply the RTOS kernels to be deployed on embedded projects. It is also possible to deploy the RTOS software over Linux and simulate the behaviour of tasks, processes and interrupts without the actual need of hardware. Such a simulation is very powerful as it helps one study and understand the workings of the RTOS, design applications and check the feasibility and even explore the various features provided by the RTOS supplier, all without having to constantly depend on a host hardware setup. This is particularly helpful during situations when access to hardware is limited (for example there being only one board and multiple developers, it is sometimes not possible for all developers to access the hardware) or when access to hardware is not possible (costs, technical issues, distance learning as is the current case).

The objective of this project is to try and execute various open source RTOS on a basic Linux deployment on either a laptop or PC. This report contains brief descriptions of each RTOS, features provided and an extensive guide on how to setup and get it running on your Linux system. It also includes examples which encompass and demonstrate the basic features such as task creation, priority definition, creating periodic tasks and synchronization between two tasks.

## 2. FreeRTOS

### 2.1 Introduction

FreeRTOS is a real time operating system distributed freely under MIT and developed in partnership with various companies. It is a very simple and easy to use RTOS, and according to the claims of the company, quite popular as well. The core kernel of the RTOS is completely contained on only 3 C files and the images generated by each build is generally quite small around 6-12 kB. This makes it quite attractive for use in small boards which may not have much storage capacity. It is constantly under revision and has new releases published every year with possibly newer features.

It also has a very active and helpful forum through which it is possible to discuss and troubleshoot problems with various other developers across the world. The environment setup is possibly the quickest among all the OS that will be described as a part of this project.

The boards supported by FreeRTOS are boards from ATMEL, STM, ARM, NXP and many others which can be found on their page : [https://www.freertos.org/RTOS\\_ports.html](https://www.freertos.org/RTOS_ports.html)

### 2.2 Setup

The steps below show how to download, compile and run the FreeRTOS from scratch. These steps will work provided the user has installed the basic tools on Linux (Refer Basic Installation)

- I. Download the source file from the link given below preferably into a project folder. <https://www.freertos.org/a00104.html>. It is also possible to do a git clone from the GitHub page: <https://github.com/freertos/freertos>. Run the command

```
git clone https://github.com/freertos/freertos
```

in your desired folder to download onto your computer.

- II. Download the file provided in the page : <https://www.freertos.org/FreeRTOS-simulator-for-Linux.html>. It can be accessed by clicking the text highlighted in blue as shown below.

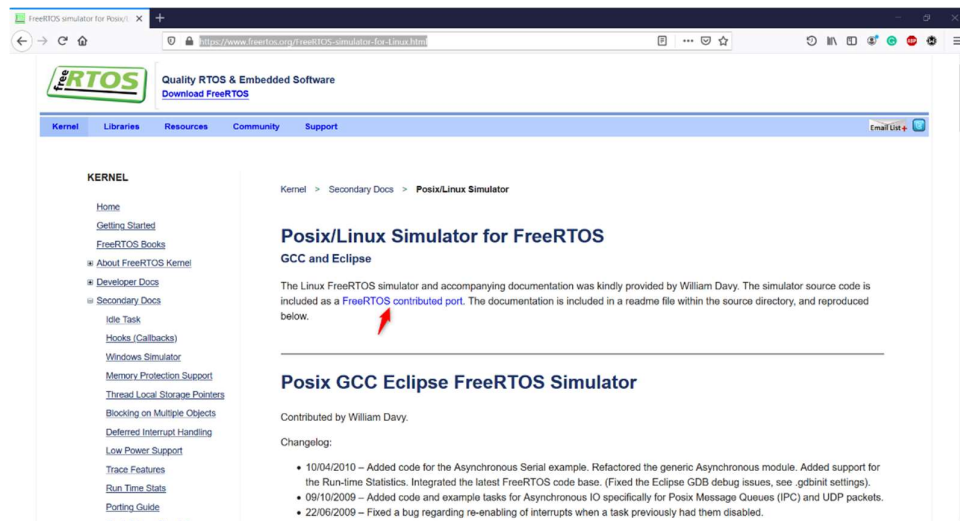


Fig:1 Link on FreeRTOS website to download the POSIX simulator.

III. Extract the contents of the zip file and copy onto

`project_folder/FreeRTOSv10.3.1/FreeRTOS/`

IV. Now navigate to `Posix_GCC_Simulator/ FreeRTOS_Posix/ Debug`

V. Open the terminal

VI. Run the command make.

VII. The auto generated make file might have an issue with linking. In the tool invocations section, the line

```
gcc -pthread -lrt -o "FreeRTOS_Posix" $(OBJJS)
$(USER_OBJJS) $(LIBS)
```

Must be changed to

```
gcc -pthread -o"FreeRTOS_Posix" $(OBJJS) $(USER_OBJJS)
$(LIBS) -lrt
```

The make file must generally not be modified since it is automatically generated and given by the creator of this Posix simulator. When using eclipse, make sure to uncheck the generate make files automatically.

VIII. Run

```
$ ./FreeRTOS
```

- IX. This should now execute the RTOS and run the demo application which consists of multiple examples.
- X. Navigate one folder backwards into / `FreeRTOS_Posix`.
- XI. Here we find a `main.c` file and it is this file which is compiled to generate the final executable.
- XII. Upon opening the `main.c` file we see that there are multiple demo functions, headers and definitions. We can delete all the unwanted demo files (except for the few functions which are needed, this can be cross checked with the final code given).
- XIII. Now try printing “Hello World”. Add the statement to print “Hello World”. Go back into the debug folder and run the following commands in succession.

```
$ make clean
```

```
$ make
```

```
$ ./FreeRTOS
```

- XIV. This should print Hello World on your terminal (along with other lines).
- XV. Now you can start writing your code.
- XVI. The makefile is already given by the creators of this demo example and should not be modified.

## 2.3 Task Creation

This section walks you through the basic task creation API used in FreeRTOS. The header `task.h` has to be added to the C file which tries to access the API.

The function `xTaskCreate` is used for creating new tasks and adding them to the scheduling queue.

It takes 6 arguments and they are described below:

- The first argument mentions the entry point for the task, i.e. here the name of the function which the task executes is expected.
- The second argument is the name in string format
- The third argument is the stack size for the task. We set this as `configMINIMAL_STACK_SIZE`.
- The fourth argument is any parameter which is to be passed into the task. Set to `NULL` since no parameters are passed.
- The fifth parameter indicates priority. It must be mentioned as `tskIDLE_PRIORITY + a positive value` (i.e. the priority is set above the idle task priority). Higher the value, greater the priority
- The last parameter is for a task handle is needed. This is kept as `NULL`.

```
Ex: xTaskCreate(periodicTaskCreate, "test1",  
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, NULL);
```

This creates a task with name “test1”, which is allocated the minimal stack size, has a priority of +2 higher than the idle task and enters from the function “periodicTaskCreate”.

If a higher priority task is created from a lower priority task, then the execution jumps onto the task with higher priority immediately after the execution of this statement (provided pre-emptive scheduling is enabled, which it is by default).



## 2.4 Starting the scheduler

The function `vTaskStartScheduler()` is used to call the RTOS scheduler. After invoking this function the RTOS kernel has control over the tasks and their execution. Calling this task creates an idle task as well. The header `task.h` needs to be included.

This function is written generally **at the end** of an initialization function after all the basic tasks have been created, configurations done, data structures initialized etc, as once this function is successfully executed, the lines following this statement is not executed as the control now goes to the RTOS kernel.

## 2.5 Creating a periodic task

In the previous sections we saw how to create a basic task and to call the scheduler. Now we will see how we can create a task which is scheduled periodically. There are multiple ways (including use of timers) to achieve this. This method uses a task which runs an infinite loop, and within this infinite loop the second task is periodically created. The functionality is explained below.

A basic task is created, let us say that the function of the task is “periodicTaskCreate”. Two important parameters are defined in this task, the first being `xLastWakeTime` and the second being `xFrequency`. The `xLastWakeTime` records the last time at which the task was active and the `xFrequency` determines the gap between two activations. Initially the `xLastWakeTime` takes the current tick (given by the function `xTaskGetTickCount()`).

The function `vTaskDelayUntil` puts the task into blocking mode for a certain “delay duration”. This function takes the parameters `xLastWakeTime` and `xFrequency`. These mention the time in ms, therefore a value of 100 means 100ms. After the delay period has elapsed, the function is taken out of block and continues execution. The `xLastWakeTime` is also updated in the `vTaskDelayUntil` function and therefore it is not necessary to further update this value. During the active period, this task creates the new task and goes back into the blocking wait. Since this whole section is in a `while(1)` loop, and since the delay is always the same (constant), it now creates a new task after the same period.

The code is as shown below:

```
portTickType xLastWakeTime;

const portTickType xFrequency = 500;

xLastWakeTime = xTaskGetTickCount();

while(1)
{

    vTaskDelayUntil( &xLastWakeTime, xFrequency );

    xTaskCreate( periodicTask, "test3",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, NULL);

}
```

## 2.6 Semaphore creation and task synchronization

Semaphores are used for resource management and task synchronization. In this example we see how to achieve synchronization between two tasks, the first one being active periodically and the second one being activated only if a certain criterion is met. The best way to imagine this situation is by assuming the first task is a periodic task which collects data from a sensor and the second one is an aperiodic task which only performs its function if the data from the sensor is a certain value (For example, if the sensor is a ranging sensor like a RADAR, then we can imagine the aperiodic function as a “BRAKE” function which is only activated when the range value is lower than a threshold to prevent crashing). For the use of all below mentioned APIs, the header `semphr.h` must be included.

The function `xSemaphoreCreateCounting` is used to create a semaphore. It takes two parameters, the first one being the max count, i.e. the maximum number of times the semaphore can “take” i.e. be given (This is explained further down below) and the second one is the initial count, generally set as 0.

This function returns a `xSemaphoreHandle` which must be stored in a variable if we wish to access the semaphore further down the line. Hence the return value is quite

important.

The two basic functions possible on the semaphore are the `xSemaphoreGive` and `xSemaphoreTake`.

Each time we execute an `xSemaphoreGive` function, the internal count held by the semaphore is incremented. This is what the earlier statement “The maximum number of times the semaphore can take” indicates. It is the maximum number of times the `xSemaphoreGive` function can be called with reference to a certain semaphore (assuming that the semaphore isn’t taken in between). Executing further `xSemaphoreGive` functions when the maximum count is reached results in no change and even signals an error via return. This function is taking the semaphore handle as an argument.

Each time we execute an `xSemaphoreTake` function, the internal count held by the semaphore is decremented. This function returns true (`pdTRUE`) or false(`pdFALSE`). The return value is `pdTRUE` if and only if there is an object to take, i.e. if and only if the internal count is a value greater than 0. This function takes the semaphore handle and a tick time which indicates how long to wait for the semaphore (block time) as the arguments. Block time of zero indicates that the task is polling for the semaphore and a block time of `portMAX_DELAY` will block the task indefinitely.

In the example shown below, the periodic task gives to the semaphore and the aperiodic task, takes from when available and performs an action.

```
static void periodicTask(void)

{
    static int i = 0;

    i = rand()%100;

    if(i<30)

        xSemaphoreGive(xSemaphore);

    printf("Periodic = %d\n",i);

    vTaskDelete( NULL );
```

```

}

static void aperiodicTask(void)

{
    while(1)
    {
        if(xSemaphoreTake(xSemaphore, (portTickType)0) ==
pdTRUE)

            printf("Aperiodic task\n");
    }

    vTaskDelete( NULL );
}

```

Here we simulate the sensor value via a randomly generated value (between 0 and 100). This is generated in the periodic function (this is equivalent to reading the data). Each time this value is lesser than 30, the periodic function gives to the semaphore. Simultaneously, the aperiodic task is waiting for the semaphore, i.e. it is polling to check if the semaphore has an object/value. As soon as it gets a value, the semaphore is taken the "AperiodicTask" is printed onto the console. This indicates some processing. The program is run using the make and the bash commands as before.

```

Periodic = 83
Periodic = 86
Periodic = 77
Periodic = 15
Aperiodic task
Periodic = 93
Periodic = 35
Periodic = 86
Periodic = 92
Periodic = 49
Periodic = 21
Aperiodic task
Periodic = 42
Periodic = 27
Aperiodic task
Periodic = 90
Periodic = 59
Periodic = 63
Periodic = 26
Aperiodic task
Periodic = 40
Periodic = 26
Aperiodic task
Periodic = 72
Periodic = 36
Periodic = 11
Aperiodic task
Periodic = 68
Periodic = 67
Periodic = 29
Aperiodic task

```

Fig:2 Result of synchronization using semaphores

In the result above, we see that the periodic task runs every 500ms and fetches a value given by the random value generator (inbuild C function). Whenever the value is lower than 30, it increases the semaphore count which is now available for the aperiodic task and it begins its execution.

### 3. ChibiOS

#### 3.1 Introduction

ChibiOS is fast real time operating system which is open source (GPL3 General Public license) developed by Giovanni Di Sirio. ChibiOS/RT is designed for embedded applications running on different architectures such like ARM, Atmel AVR, Freescale, Renesas and many other also it supports 8,16,32-bit microcontrollers. ChibiOS stands for Small Operating System (Chibi is a Japanese word which means small).

Key features of ChibiOS/RT : (Detailed features are explained at wiki ChibiOS)

- Supports multithreading and pre-emption.
- **Small kernel size. (kernel size minimum ranges from 1.2 Kb to a maximum of 5.5 Kb).**
- **Can create Mutexes, Semaphores, and Messages.**
- **Can support systems with ROM size about 6Kb.**

- **Hardware Abstraction Layer with support for ADC, CAN, SPI, PWM etc.**
- **Almost totally written in C with little ASM code required for ports.**

To develop an application using ChibiOS we need proper toolchains.

Toolchain consists of set of distinct software development tools linked together. So, we need Specific embedded libraries able to work on our microcontroller, Compiler able to generate code which will run most likely on a different architecture this is known as a cross-compiler, Flasher to flash the binary into the target, Debugger to test code at run-time. All those things are provided all together in ChibiStudio.

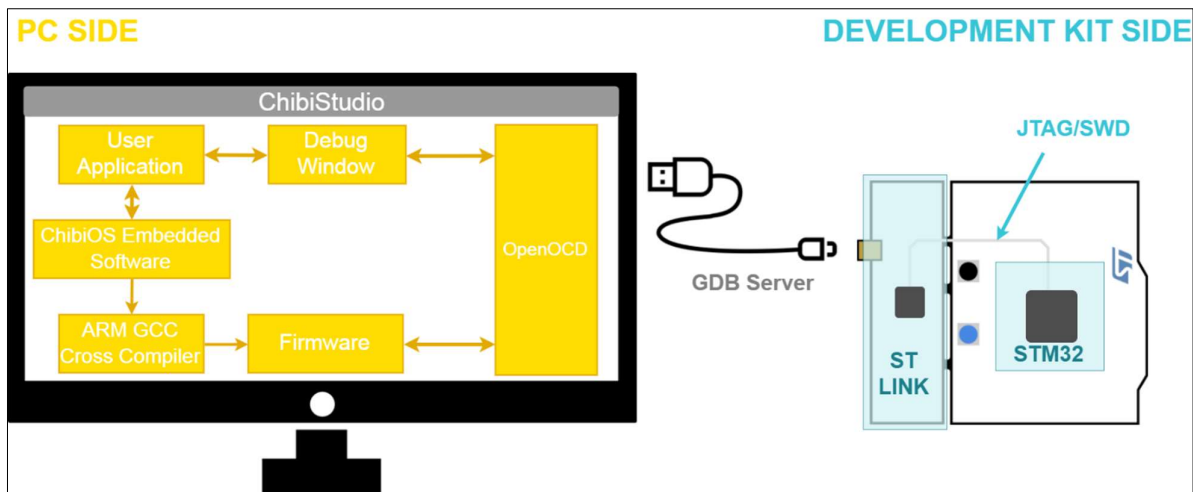
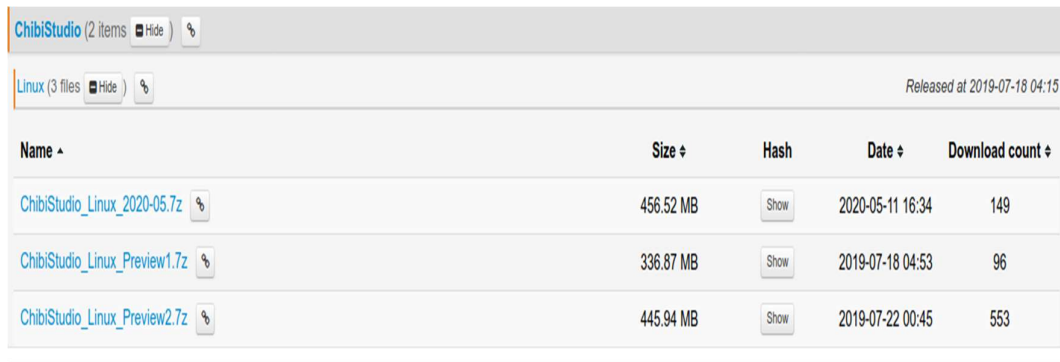


Fig3: Connection between Chibistudio and Target. Taken from [Source](#)

On the PC side, we have the toolchain which contains the ChibiOS Embedded Software and the user application. It is compiled through the ARM GCC Cross Compiler generating a firmware. On the Development kit side, we have the STM32 connected through the JTAG to the ST-Link debugger. The two sides are interconnected by the debugger which is composed of a hardware layer (STLink + Cable) plus a software layer (OpenOCD and GDB Server). Using this interconnection is possible to flash the firmware on the STM32. The target may change depending upon the application.

### 3.2 Setup for ChibiOS/RT

- Download the Latest ChibiOS source files from the below web page. <https://osdn.net/projects/chibios/releases/> . You will find there are different zip files available. We have to choose for Linux system with latest version. Refer following figure to download correct files for Linux.



Name	Size	Hash	Date	Download count
ChibiStudio_Linux_2020-05.7z	456.52 MB	Show	2020-05-11 16:34	149
ChibiStudio_Linux_Preview1.7z	336.87 MB	Show	2019-07-18 04:53	96
ChibiStudio_Linux_Preview2.7z	445.94 MB	Show	2019-07-22 00:45	553

Fig 4: Files to download for Linux

- Create a Directory to extract the files. In our case we have created directory ChibiOS and copy all the files in that directory.
- After extracting files, you will observe there are many versions of ChibiOS are present. We will work with **chibios\_stable-20.3. x**. Go through the **readme** document which will give information about product also will give you steps for installation.
- Now, we will try to look into demos. We will take demos from chibios\_stable-20.3. x. We will try to run basic demo example.
- We will run **RT-Posix-Simulator** project which is present inside folder chibios\_stable-20.3.x/demos/various. We will not try to run all the other demos which is present under folders such as AVR, ARM because ChibiOS demos are meant to run on Hardware, those are not Linux executable files.
- Now we will build this project. We use following commands to build and execute the project. To build the current project change current directory to that demo/ project location where the C files and makefile is present. Now after that run the following command to build.

`$ make`

- If no error occurs, then inside the build folder you will observe executable file name '**ch**'
- Then go inside build folder using `$ cd build` and use following commands to execute the program.

`$ chmod +x ch`

`$/ch`

- The demo listens on the two serial ports, when a connection is detected a thread. You can develop your ChibiOS/RT application using this demo as a simulator.
- We can run the project from Eclipse. You can use launcher file '**ChibiStudio-GCC9.2.1.sh**' to start the eclipse. Inside that file environment variables for ChibiOS/RT and ChibiStudio and another toolchain is set. And then eclipse will be launch.
- After starting eclipse, we have to import existing project to current workspace. Along with that we have to make few other changes to configure environment. The detail explanation about this steps is given [Configure Eclipse](#) page.
- To build a project right click on it and select **Build Project**. Another way is to click on it and press the hammer button in the tool bar. If the procedure successfully ends a new folder into the project will appear it contains the output of make process as bin, hex, elf and other formats. Now we can click on 'run' button to run the application.
- Advantage from working on Eclipse:
  1. Debugging will be very easy.
  1. Syntax errors can easily identify.
  2. Build and Run will be possible with one click.

### 3.3 Task Creation

In this section we will discuss about creation of basic Task in ChibiOS. In ChibiOS/RT a **thread** is actually a Task (like sequence of instruction). ChibiOS also supports multi-threading.



Creation of thread includes assigning a priority to a thread and a working area. The working area is a piece of memory dedicated to our thread and a priority level is a relative number which we can use to influence the scheduling. Priority follows a simple rule, among all the threads ready for execution, the one with the highest priority is the one being executed.

Refer different API related to threads from [ChibiOS/Threads](#) webpage.

We will use them for developing our application. This document is the ultimate reference manual for the ChibiOS. So, it will help for developing application.

### 3.4 Creating a periodic task

Now in this section we will learn about how to create periodic task and how context switching is happened in the ChibiOS.

We will create a simple application in which we will create a periodic task which will be activate at every 500ms and prints "Periodic Task" on console. Also create a static variable and increment the value after every activation of Task. Print this value also.

All tasks are the same in ChibiOS, you can do periodic or aperiodic operations inside thread **functions. There are examples for periodic tasks in the user manual here** [ChibiOS/Periodic Tasks](#). **Look at the 3 solutions for "fixed intervals".**

**Steps for design of application are as follows.**

#### 3.4.1 Make a Project Folder

As you can see in above example we have successfully execute the demo RT-Posix-Simulator from ChibiOS-stable-20.3.x. So in this application also there is no changes in the make file (add the write the make file) and configuration files hence we can use similar files in our project. We will copy and paste the same project and change the name to 'first\_application'. We will write code into **main .c** file. This is the only file we will be modify for this application.

#### 3.4.2 Adding headers

Adding headers files will allow us to access all the API's related to ChibiOS into our application. For this application we required following headers to be included.

```
#include "ch.h"
```

```
#include "hal.h"
#include <stdio.h>
```

### 3.4.3 Writing Make File

Makefile is a script file which contains a series of directives used by GCC make to build our code. Basically, it contains paths of imported source files, name of executable file, Licensing files, compiler information and settings.

[GNU Make Manual.](#)

### 3.4.4 Allocating a working area to thread.

Allocates a working area where **waThread1** is its identifier and **128** its size expressed in Bytes.

```
static THD_WORKING_AREA(waThread1, 128);
```

### 3.4.5 Declaring a thread function.

In this case, we are declaring a working area named **waThread1** having size 128 bytes. We are also declaring a function named **Thread1**. Inside the function we will call a function **periodicTask()** then suspends the thread operation for 500ms.

In **PeriodicTask()** function we will write a program to print "Periodic Task" and static variable value on console.

```
static THD_WORKING_AREA(waThread1, 128);
static THD_FUNCTION(Thread1, arg) {
    (void) arg;
    chRegSetThreadName("Taks1");
    while (true) {
        periodicTask();
        chThdSleepMilliseconds(500);
    }
}
```

## 3.5 Task creation and scheduling

- When the application starts the two functions (**halInit()** and **chSysInit()**) perform the system initialization: while **halInit()** is an API of ChibiOS/HAL and initializes the HAL subsystem, **chSysInit()** is an API of ChibiOS/RT and initializes the kernel. Note that every application based on ChibiOS/RT and ChibiOS/HAL begins in the same way.

- After `chSysInit()` the `main()` becomes a thread itself and another thread named `idle` is created: `idle` is a dummy thread which runs when all the other threads are not ready to run.
- It is important that `hallInit()` and `chSysInit()` are executed at the beginning of `main()`. Using any ChibiOS API before them will have an unwanted behavior probably causing a system crash.
- We can create 'Thread1' inside the main using the API `chThdCreateStatic ()`. Below example will give the illustration of usage of this API.
- For ex. **`chThdCreateStatic (waThread1, sizeof(waThread1), NORMALPRIO, Thread1, NULL);`**

This API is commonly used and requires 5 arguments.

1. **`waThread1`** - A pointer to the working area (`waThread1`, the one we have declared before).
1. **`sizeof(waThread1)`** - The size of the working area.
2. **`NORMALPRIO`** - The priority of thread. The priority. In ChibiOS priorities are integer in a range which starts from `LOWPRIO` up to `HIGHPRIO`. The main thread is started at the `NORMALPRIO` priority level which is in the middle of the whole range. Among all the threads ready for execution, the one with the highest priority is the one being executed, no exceptions to this rule.
3. **`Thread1`** - A function to execute as a thread (`Thread1`, which has been declared before).
4. **`NULL`** - A parameter which will be passed to the `Thread1` function (in this case we are passing nothing).

- Every thread must have a sleep or a suspending function inside its loop. In this way the CPU ownership is switched among threads and the scheduling can proceed. ChibiOS/RT offers different sleep functions but commonly used API is **chThdSleepMilliseconds(<time>)**. This function suspends a thread for <time> milliseconds, where the argument is an unsigned integer.

For ex. **chThdSleepMilliseconds (500);** This function will suspend the current thread for 500 milliseconds.

- In our case we have used this API in the Thread function, after calling periodicTask() function. We suspend current thread for 500ms. Pre-emption happens when another thread with a highest priority becomes ready for the execution. In this case there is a context switch from the lower priority thread to higher priority thread. Look at the following code and you will get to know about Thread creation and how to use chThdSleepMilliseconds in code.

### 3.5.1 Explanation of Scheduling and pre-emption of threads

- In this case we have three threads. Ordering them by decreasing priority they are Thread1, main and idle. The last one is a thread always ready with the lowest priority which is executed only when other thread are not ready. As its name suggest, it is an empty thread which does nothing waiting for an event.
- Threads are ordered by priority on the y-axis, on the x-axis there is time.
- The coloured rectangles represent the current thread. A grey arrow represents that related thread is sleeping, otherwise green arrow represent time in which a thread is ready to be executed but waits other threads with higher priority. Now, we can explain timing flow.
- The application enters in main (). When chsylvnit() is executed main() becomes current thread. It runs until the creation of Thread1.
- When Thread1 has been created both main and Thread1 are Ready but Thread1 has highest Priority. So main () is pre-empted. Thread1 then calls periodicTask () function which prints “Periodic Task” and value of variable ‘i’ on console and goes sleep for 500ms.

- Now main is resumed and enters into its loop then goes sleep for 800ms.
- Both main and Thread1 are suspended and idle is the only one to be ready then is executed.
- Thread1 becomes ready and pre-empts idle. Then then calls periodicTask() function which prints “Periodic Task” on console and goes sleep for 500ms. These repeats as we are executing all Threads in infinite loop.
- Refer below figure which will be explained about timing flow.

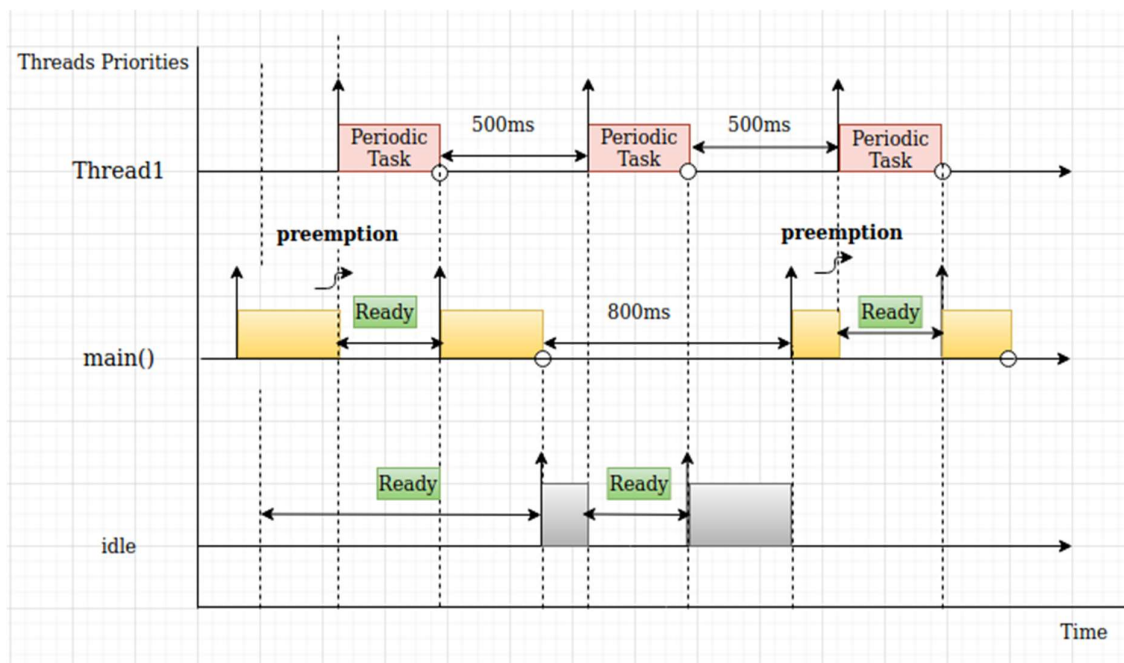


Fig 5: Timing flow of periodic Task

### 3.5.2 Steps to execute the program.

- Open terminal. Change current directory to project location.
- Now run command `$ make clean`. This will clean the previous build if any.
- Then we have build it again. So, run the command `$make`

- Look for errors. If you follow exact steps then there is very less possibilities to have errors for toolchain. If you have syntax issues then resolve it. And build it again.
- Now change current directory to build folder. `$ cd build`
- If you see executable with file name 'ch' is created. Now to we will try to run executable.
- Give permission to run. Use command `$ chmod +x ch`.
- Run executable. `$ ./ch`

### 3.5.3 Results.

When you run the application then on the terminal you will see message "Periodic Task" which will be displayed after every 500ms.

```
varad@varad-HP-Pavillon-Notebook:~/ChibiStudio/chibios_stable-20.3.x/demos/various/RT-Posix-Simulator/build$ chmod +x ch
varad@varad-HP-Pavillon-Notebook:~/ChibiStudio/chibios_stable-20.3.x/demos/various/RT-Posix-Simulator/build$ ./ch
ChibiOS/RT simulator (Linux)
Periodic Task = 1
Periodic Task = 2
Periodic Task = 3
Periodic Task = 4
Periodic Task = 5
Periodic Task = 6
Periodic Task = 7
Periodic Task = 8
Periodic Task = 9
Periodic Task = 10
Periodic Task = 11
Periodic Task = 12
Periodic Task = 13
Periodic Task = 14
Periodic Task = 15
Periodic Task = 16
Periodic Task = 17
Periodic Task = 18
Periodic Task = 19
Periodic Task = 20
Periodic Task = 21
Periodic Task = 22
Periodic Task = 23
```

Fig7: Result of Periodic Task application

### 3.6 Semaphore creation and task synchronization

- Semaphore is an integer variable which is used in mutual exclusive manner by various threads which are sharing some common resource in order to achieve synchronization. Basically, it is signalling mechanism which allows several threads to access shared resources. A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource

(decrementing the semaphore) and can signal that they have finished using the resource (incrementing the semaphore).

- In this example we see how to achieve synchronization between two tasks, the first one being active periodically and the second one being activated only if a certain criterion is met.
- Here we simulate the sensor value via a randomly generated value (between 0 and 100). This is generated in the periodic function (this is equivalent to reading the data). Each time this value is lesser than 30, the periodic Task will call the aperiodic Task. And aperiodic Task will execute and suspends. We use semaphores to use the synchronization for this. Please follow below steps to design this task.

### 3.6.1 Adding header file

- In addition to headers which we have included in previous task we will add header files performing rand () operation. This function will generate random value between 0 to 100

```
# include <stdlib.h>
#include <time.h>
```

### 3.6.2 Writing a Make file.

- In this application Make file will be not changed. Because this is the extension of our first application only. So, need to change anything in that.

### 3.6.3 Create aperiodic Task

- Now we will create aperiodic task which will activate from periodic task. The steps for creating aperiodic task, Task function will be same. Only change is the name of the thread will be changed. Check the below the code.
- `chThdCreateStatic (waThread2, sizeof(waThread2), NORMALPRIO, Thread2, NULL);` - This will create Thread2 (which we refer as our aperiodic Task). You will add this line in main function just below where you have previously created the Thread 1.

- Now writing Thread function also will be same as above steps.

### 3.6.4 Semaphore

- After this we will see how to can use semaphores. Semaphore is a mechanism that can be used to provide synchronization of tasks.
- There are two types semaphores. One is counting semaphore and other is binary semaphore.
- **Counting Semaphore** uses a count that helps task to be acquired or released numerous times. If the initial count = 0, the counting semaphore should be created in the unavailable state. However, If the count is > 0, the semaphore is created in the available state, and the number of tokens it has equals to its count.
- The **binary semaphores** are quite similar to counting **semaphores**, but their value is restricted to 0 and 1. In this type of **semaphore**, the wait operation works only if **semaphore = 1**, and the signal operation succeeds when **semaphore= 0**. It is easy to implement than **counting semaphores**
- When a task attempts to acquire a semaphore that is unavailable, the semaphore places the task onto a **wait queue and puts the task to sleep**. The processor is then free to execute other code. When the semaphore becomes available, one of the tasks on the wait queue is awakened so that it can then acquire the semaphore.
- So in our application we will use binary semaphore.
- First, we will create an object which is pointing to a pointer **semaphore\_t** structure.
- We declare **sem object** which **semaphore\_t** structure as following. This should be defined at the beginning of code.

```
static binary_semaphore_t sem;
```

- Now we will Initializes a semaphore with the specified counter value. We can initialize in **main function** by using API `chSemObjectInit()` API. The usage of this API is explained below. It has two input argument.



```
void chSemObjectInit (semaphore_t * bsp, bool taken );
```

sp - pointer to a semaphore\_t structure

taken- initial state of the binary semaphore.

**False:** the initial state is not taken.

**True:** the initial state is taken.

- For example, in our application we have initialize semaphore like

```
chBSemObjectInit (sem, true);
```

- After that in periodic task signals to the semaphore and the aperiodic task, waits till it is available and performs an action. This is done using two API's. **chBSemSignal(binary\_semaphore\_t \* bsp)** and another AP **chSemWait(binary\_semaphore\_t \* bsp)**.
- In peridoc task we used **chBSemSignal(&sem)**. The signal operation is executed when a process takes exit from the critical section.
- Now in our aperiodic Task we will use **chSemWait()** to check if it is available for performing the action.
- This function Performs a wait operation on a semaphore. Input to this function is pointer to a **binary\_semaphore\_t** structure. This gives output as '**MSG\_OK**' if the thread has not stopped on the semaphore or the semaphore has been signalled. And '**MSG\_RESET**' if the semaphore has been reset
- Example below gives complete design of the application.

```
static void periodicTask(void);  
#define SHELL_WA_SIZE THD_WORKING_AREA_SIZE(1024)  
static binary_semaphore_t sem;  
  
static void periodicTask(void)  
{  
    static int i = 0;  
    i = rand()%100;
```

```

        if(i<30)
            chBSemSignalI( &sem );
        printf("Periodic = %d\n",i);
    }

    /*
    * Periodic Task thread, times are in milliseconds.
    */

    static THD_WORKING_AREA(waThread1, 128);
    static THD_FUNCTION(Thread1, arg) {

        (void)arg;
        chRegSetThreadName("Thread1");
        while (true) {

            periodicTask();

            chThdSleepMilliseconds(2000);
        }
    }

    /*
    * aperiodic Task thread, times are in milliseconds.
    */

    static THD_WORKING_AREA(waThread2, 128);
    static THD_FUNCTION(Thread2, arg) {

        (void)arg;
        chRegSetThreadName("Thread2");
        while (true) {

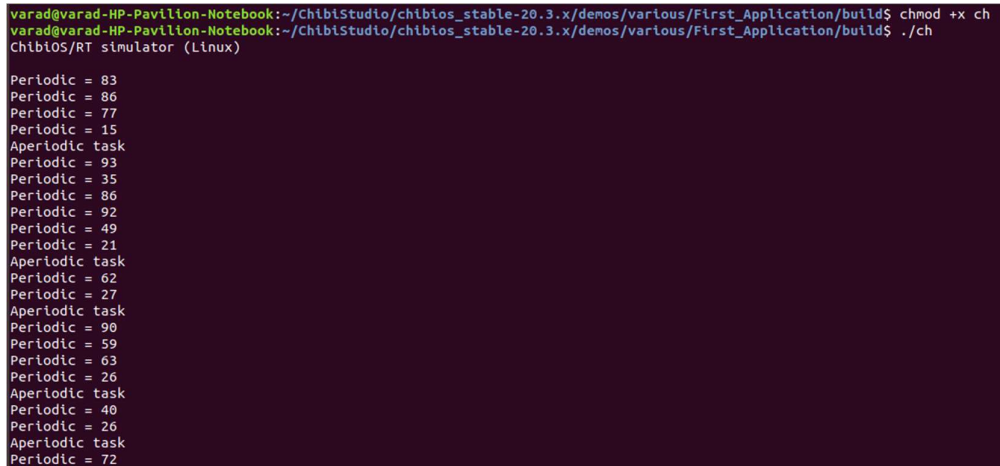
            if( chBSemWait( &sem ) == MSG_OK )
            {
                printf("Aperiodic task\n");
            }

        }
    }
}

```

### 3.6.5 Result

This application displays '**periodic Task**' whenever periodicTask activates. And as soon as the condition for activation of aperiodic task met and semaphore is available then it will print '**Aperiodic Task**'.



```
varad@varad-HP-Pavilion-Notebook:~/ChibiStudio/chibios_stable-20.3.x/demos/variouse/First_Application/build$ chmod +x ch
varad@varad-HP-Pavilion-Notebook:~/ChibiStudio/chibios_stable-20.3.x/demos/variouse/First_Application/build$ ./ch
ChibiOS/RT simulator (Linux)
Periodic = 83
Periodic = 86
Periodic = 77
Periodic = 15
Aperiodic task
Periodic = 93
Periodic = 35
Periodic = 86
Periodic = 92
Periodic = 49
Periodic = 21
Aperiodic task
Periodic = 62
Periodic = 27
Aperiodic task
Periodic = 90
Periodic = 59
Periodic = 63
Periodic = 26
Aperiodic task
Periodic = 40
Periodic = 26
Aperiodic task
Periodic = 72
```

Fig8: Result of task with semaphores

## 4. RTEMS

### 4.1 Introduction

RTEMS is an open source RTOS that supports open standard application programming interfaces such as POSIX. It is used in space flight, medical, networking and many more embedded development applications. RTEMS supports processor architectures including ARM, Atmel AVR PowerPC, Intel, Blackfin, MIPS, Microblaze, Renesas and more. It is free open source software. Some key features about RTEMS are

- POSIX 1003.1b API including threads.
- TCP/IP including BSD Sockets.
- GNU Toolset Supports Multiple Language Standards. Multitasking capabilities.
- Optional rate-monotonic scheduling. Inter-task communication and synchronization.
- Priority inheritance.
- Dynamic memory allocation.

- High level of user configurability.
- Portable to many target environments.
- High performance port of FreeBSD TCP/IP stack.

Now we will see the detailed installation guide for RTEMS.

## 4.2 Steps for installation

You need to perform some basic preparation to get started with RTEMS development. You need tools from your host's operating system to build the RTEMS tool suite from source.

The RTEMS tools you build are used to build the Board Support Package (BSP) libraries for your target hardware from source. The BSP libraries contain the RTEMS operating system.

### 4.2.1 Setting up environment

- BSP Build will require pax package for RTEMS configuration. this package is not installed, by default, on many Linux distributions. Also, you need a native C, C++, and Python development environment. So, if it is not installed on your machine then use below commands to install it.

```
$ sudo apt-get build-dep build-essential gcc-defaults g++ gdb git \unzip pax bison flex
texinfo unzip python3-dev libpython-dev \libncurses5-dev zlib1g-dev
```

- It is likely necessary that you will have to enable the Ubuntu Source Repositories. Users have suggested the following web pages which have instructions:

<https://askubuntu.com/questions/158871/how-do-i-enable-the-source-code-repositories/158872>

- If you are new to RTEMS and you are looking to try RTEMS then the best suited Board Support Package (BSP) is the SPARC ERC32 (erc32). The SPARC ERC32 BSP has a robust simulator that runs the example and test executables on your host computer. Next sections will give us brief idea about how to build the erc32 BSP and run RTEMS tests executables in the simulator. The ERC32 BSP is a SPARC architecture BSP so the tool suite name is **sparc-rtems5**.

#### 4.2.2 Select installation prefix.

- You have to select a prefix for your installation. You will build and install the RTEMS tool suite, an RTEMS kernel for a BSP, and you may build and install third party libraries. You can build all the parts as a stack with a single prefix or you can separate various parts by providing different prefixes to each part as it is built. Using separate prefixes is for experienced RTEMS users.
- Do not select a prefix that is under the top of any of the source trees. The prefix collects the install output of the various build steps you take in this guide and need to be kept separate from the sources used.
- It is strongly recommended to run the RSB as a normal user and not with root privileges (also known as super user or Administrator). You have to make sure that your normal user has sufficient privileges to create files and directories under the prefix. You can choose a prefix in our home directory, e.g. `$HOME/rtems/5` or with creating project folder like `$HOME/development/rtems/5`.
- For our installation, we will choose `$HOME/quick-start/rtems/5` for the RTEMS tool suite prefix.

#### 4.2.3 Getting Source code

- You need the RTEMS Source Builder (RSB) to work with RTEMS and we prefer you use a released version. A released version of the RSB downloads all source code from the RTEMS servers.
- We can directly git clone the source files into `$HOME/quick-start/src` path.
- Use below instructions to get the source files

```
mkdir -p $HOME/quick-start/src
cd $HOME/quick-start/src
git clone git://git.rtems.org/rtems-source-builder.git rsb
git clone git://git.rtems.org/rtems.git
```

#### 4.2.4 Install Tool suite.

- You have chosen an installation prefix, the BSP to build, the tool's architecture and prepared the source for the RSB in the previous sections. We have chosen `$HOME/quick-`

`start/rtems/5` as the installation prefix, the `erc32` BSP and the `SPARC` architecture name of `sparc-rtems5`, and unpacked the RSB source in `$HOME/quick-start/src`. The tool suite for RTEMS and the RTEMS sources are connected. For example, do not use a RTEMS version 5 tool suite with RTEMS version 4.11 sources and vice versa.

- Use below instructions to build and install the tool suite for RTEMS version 5. You will see output similar to below figure.

```
$cd $HOME/quick-start/src/rsb/rtems
$../source-builder/sb-set-builder --prefix=$HOME/quick-
start/rtems/5 5/rtems-sparc
```

```
RTEMS Source Builder - Set Builder, 5.1.0
Build Set: 5/rtems-sparc
...
config: tools/rtems-binutils-2.34.cfg
package: sparc-rtems5-binutils-2.34-x86_64-freebsd12.1-1
building: sparc-rtems5-binutils-2.34-x86_64-freebsd12.1-1
sizes: sparc-rtems5-binutils-2.34-x86_64-freebsd12.1-1: 305.866MB (installed: 29.966MB)
cleaning: sparc-rtems5-binutils-2.34-x86_64-freebsd12.1-1
reporting: tools/rtems-binutils-2.34.cfg -> sparc-rtems5-binutils-2.34-x86_64-freebsd12.1-1.txt
reporting: tools/rtems-binutils-2.34.cfg -> sparc-rtems5-binutils-2.34-x86_64-freebsd12.1-1.xml
config: tools/rtems-gcc-7.5.0-newlib-fbba096.cfg
package: sparc-rtems5-gcc-7.5.0-newlib-fbba096-x86_64-freebsd12.1-1
building: sparc-rtems5-gcc-7.5.0-newlib-fbba096-x86_64-freebsd12.1-1
....
Build Sizes: usage: 5.684GB total: 1.112GB (sources: 143.803MB, patches: 21.348KB, installed 995.1MB)
Build Set: Time 0:21:35.626294
```

Fig: 8 Result after installing Tool suite for RTEMS

- Once the build has successfully completed you can check if the cross-C compiler works with the following command. The version information helps you to identify the exact sources used to build the cross compiler of your RTEMS tool suite.

```
$HOME/quick-start/rtems/5/bin/sparc-rtems5-gcc -version
```

#### 4.2.5 Bootstrap the RTEMS Sources

- Bootstrap is important that the right version of Autotools (`autoconf` and `automake`) are in your `$PATH` (`$HOME/quick-start/rtems/5/bin`) . The right version of Autotools is shipped with the RTEMS tool suite you already installed.
- Set the path to the tool suite installed under your selected prefix by using below command.

```
$export PATH=$HOME/quick-start/rtems/5/bin:$PATH
```

- Then go RTMS source folder and bootstrap the build system. Use below command to this. Result will be similar to below figure.

```
$cd $HOME/quick-start/src/rtems  
$./rtems-bootstrap
```

```
RTEMS Bootstrap, 1.0  
1/122: autoreconf: configure.ac  
2/122: autoreconf: testsuites/configure.ac  
3/122: autoreconf: testsuites/fstests/configure.ac  
4/122: autoreconf: testsuites/smptests/configure.ac  
5/122: autoreconf: testsuites/psxtests/configure.ac  
6/122: autoreconf: testsuites/mptests/configure.ac  
...  
121/122: autoreconf: c/src/lib/libbsp/lm32/milkymist/configure.ac  
122/122: autoreconf: c/src/make/configure.ac  
Bootstrap time: 0:00:46.404643
```

Fig:9 Result after bootstrap

#### 4.2.6 Build a Board Support Package (BSP)

- You are now able to build Board Support Packages (BSPs) for all architectures you have an installed RTEMS tool suite. To build applications on top of RTEMS, you have to build and install a BSP for your target hardware. We select the **erc32** BSP.
- We build the BSP in four steps. The first step is to create a build directory. It must be separate from the RTEMS source directory. We use \$HOME/quick-start/build/b-erc32. We will create the folder under above directory by using below command.

```
$mkdir -p $HOME/quick-start/build/b-erc32
```

- The second step is to set your path. Prepend the RTEMS tool suite binary directory to your \$PATH throughout the remaining steps. Run the command.

```
$export PATH=$HOME/quick-start/rtems/5/bin:"$PATH"
```

- Check your installed tools can be found by running below command. The output “found”. If not found is printed the tools are not correctly installed or the path has not been correctly

set. Check the contents of the path `$HOME/quick-start/rtems/5/bin` manually and if `sparc-rtems5-gcc` is present the path is wrong. If the file cannot be found return to Install the Tool Suite and install the tools again.

```
$command -v sparc-rtems5-gcc && echo "found" || echo "not found"
```

- The third step is to configure the BSP. There are various configuration options available. Some configuration options are BSP-specific. Run the following command. Result will be similar to below figure.

```
$cd $HOME/quick-start/build/b-erc32
$~/HOME/quick-start/src/rtems/configure \
    --prefix=$HOME/quick-start/rtems/5 \
    --enable-maintainer-mode \
    --target=sparc-rtems5 \
    --enable-rtemsbsp=erc32 \    --enable-tests
```

```
checking for gmake... gmake
checking for RTEMS Version... 5.0.0
checking build system type... x86_64-unknown-freebsd12.0
checking host system type... x86_64-unknown-freebsd12.0
checking target system type... sparc-unknown-rtems5
...
config.status: creating Makefile

target architecture: sparc.
available BSPs: erc32.
'gmake all' will build the following BSPs: erc32.
other BSPs can be built with 'gmake RTEMS_BSP="bsp1 bsp2 ..."'

config.status: creating Makefile
```

Fig: 10 Result after configuring BSP

- Fourth step is building BSP. Run the below command. Result will be similar to below figure.



```
$cd $HOME/quick-start/build/b-erc32
$Make
```

```
Configuring RTEMS_BSP=erc32
checking for gmake... gmake
checking build system type... x86_64-unknown-freebsd12.0
checking host system type... sparc-unknown-rtems5
checking rtems target cpu... sparc
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for sparc-rtems5-strip... sparc-rtems5-strip
checking for a thread-safe mkdir -p... $BASE/src/rtems/c/src/../../install-sh -c -d
checking for gawk... no
checking for mawk... no
checking for nawk... nawk
checking whether gmake sets $(MAKE)... yes
checking whether to enable maintainer-specific portions of Makefiles... yes
checking for RTEMS_BSP... erc32
checking whether CPU supports libposix... yes
configure: setting up make/custom
configure: creating make/erc32.cache
gmake[3]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
...
sparc-rtems5-gcc -mcpu=cypress -O2 -g -ffunction-sections -fdata-sections -Wall -Wmissing-prototypes
gmake[5]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32/testsuites/sptests'
gmake[4]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32/testsuites'
gmake[3]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
gmake[2]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
gmake[1]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c'
gmake[1]: Entering directory '$BASE/build/b-erc32'
gmake[1]: Nothing to be done for 'all-am'.
gmake[1]: Leaving directory '$BASE/build/b-erc32'
```

Fig: 11 Result after building BSP

- Last step is installing BSP into `$HOME/quick-start/build/b-erc32` path. Run command

```
$cd $HOME/quick-start/build/b-erc32
$make install
```

```

Making install in sparc-rtems5/c
gmake[1]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c'
Making install in .
gmake[2]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c'
gmake[3]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c'
gmake[3]: Nothing to be done for 'install-exec-am'.
gmake[3]: Nothing to be done for 'install-data-am'.
gmake[3]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c'
gmake[2]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c'
Making install in erc32
gmake[2]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
gmake[3]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
Making install-am in .
Making install-am in cpukit
gmake[4]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32/cpukit'
gmake[5]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32/cpukit'
gmake[5]: Nothing to be done for 'install-exec-am'.
$BASE/src/rtems/c/src/../../cpukit/./install-sh -c -d '$BASE/rtems/5/sparc-rtems5/erc32/lib/inc'
...
$BASE/src/rtems/make/Templates/Makefile.lib '$BASE/rtems/5/share/rtems5/make/Templates'
$BASE/src/rtems/./install-sh -c -d '$BASE/rtems/5/make/custom'
/usr/bin/install -c -m 644 $BASE/src/rtems/make/custom/default.cfg '$BASE/rtems/5/make/custom'
gmake[2]: Leaving directory '$BASE/build/b-erc32'
gmake[1]: Leaving directory '$BASE/build/b-erc32'

```

Fig:12 Result after installing BSP

If everything works correctly as mentioned above, then we have successfully installed BSP and RTEMS. Now we will see how to run application in the next section.

#### 4.2.7 Build Hello World application

- We will now create a simple Hello World application with a Git repository and using the Waf build system.
- Waf is a build automation tool designed to assist in the automatic compilation and installation of computer software. It is written in Python. We Setup the application work space then create a new Git repository, and download the Waf build system.
- The application is be created in `$HOME/quick-start/app/hello`.
- First step is creating application directory and make it as a current directory. Run the below command.

```

$mkdir -p $HOME/quick-start/app/hello
$cd $HOME/quick-start/app/hello

```

- Then next step is Download the Waf build system and set it to executable. For this we will `curl`. So first we will install curl if you haven't installed already.

```
$ sudo apt install curl
$curl https://waf.io/waf-2.0.19 > waf
$chmod +x waf
```

- Next step is initializing new Git repository git repository and add RTEMS Waf support as a Git sub-module and initialise it. Use below commands.

```
$git init
$git submodule add git://git.rtems.org/rtems_waf.git
rtems_waf
```

- After completing above this we will create application source files. We require three types of file. First file init.c that configures RTEMS. Second is hello.c which is application source file. Third file is wscript. We will create this files in path \$HOME/quick-start/app/hello. We will be following commands to create

```
$gedit init.c
$gedit hello.c
$gedit wscript
```

- Now, as mentioned earlier 'init.c' file configures the RTEMS. The same with other types of operating systems, configuration file for RTEMS contains number of macro definitions. These macro definitions for RTEMS related with global variables and constants such as RTEMS configuration table, CPU dependency information table, the system initialization task table and user utilization task lists. 'confdefs.h' includes operating system configuration templates which can be pre compiled. So, we will add following simple RTEMS configurations to our file.

```
/*
 * Simple RTEMS configuration
 */

#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
```

```

#define CONFIGURE_UNLIMITED_OBJECTS
#define CONFIGURE_UNIFIED_WORK_AREAS

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

#define CONFIGURE_INIT

#include <rtems/confdefs.h>

```

- After this step we will write our main application. First step is mentioning correct header files. ‘#include <rtems.h>’ is to access API’s for RTEMS in main application. ‘#include <stdlib.h>’, and ‘#include <stdio.h>’ are general purpose standard library of C programming language. Next step is creating initialization task. Then we just write the line for printing “Hello world” on console. Add following lines into your code.

```

/*
 * Hello world example
 */
#include <rtems.h>
#include <stdlib.h>
#include <stdio.h>

rtems_task Init(
    rtems_task_argument ignored
)
{
    printf ("\nHello World\n");
    exit( 0 );
}

```

- Last step is writing wscript. Basically, this is similar to writing Make file, linking and compiling the program. We define executable file name, compiler required and source files for the application. Apart from this it contains functions which are necessary for RTEMS initialization. Add the below lines in the wscript file.

```

#
# Hello world Waf script
#

```

```

from __future__ import print_function

rtems_version = "5"

try:
    import rtems_waf.rtems as rtems
except:
    print ('error: no rtems_waf git submodule')
    import sys
    sys.exit(1)

def init(ctx):
    rtems.init(ctx, version = rtems_version, long_commands =
True)

def bsp_configure(conf, arch_bsp):
    # Add BSP specific configuration checks
    pass

def options(opt):
    rtems.options(opt)

def configure(conf):
    rtems.configure(conf, bsp_configure = bsp_configure)

def build(bld):
    rtems.build(bld)

    bld(features = 'c cprogram',
        target = 'hello.exe',
        cflags = '-g -O2',
        source = ['hello.c',
            'init.c'])

```

- We have completed writing into our source files. Now we will Configure the application using Waf's configure command. The output will be something close to below figure.

```

$ ./waf configure --rtems=$HOME/quick-start/rtems/5 --rtems-
bsp=sparc/erc32

```

```

Setting top to                : $BASE/app/hello
Setting out to                : $BASE/app/hello/build
RTMS Version                  : 5
Architectures                 : sparc-rtems5
Board Support Package (BSP)   : sparc-rtems5-erc32
Show commands                 : no
Long commands                 : no
Checking for program 'sparc-rtems5-gcc' : $BASE/rtems/5/bin/sparc-rtems5-gcc
Checking for program 'sparc-rtems5-g++' : $BASE/rtems/5/bin/sparc-rtems5-g++
Checking for program 'sparc-rtems5-gcc' : $BASE/rtems/5/bin/sparc-rtems5-gcc
Checking for program 'sparc-rtems5-ld'  : $BASE/rtems/5/bin/sparc-rtems5-ld
Checking for program 'sparc-rtems5-ar'  : $BASE/rtems/5/bin/sparc-rtems5-ar
Checking for program 'sparc-rtems5-nm'  : $BASE/rtems/5/bin/sparc-rtems5-nm
Checking for program 'sparc-rtems5-objdump' : $BASE/rtems/5/bin/sparc-rtems5-objdump
Checking for program 'sparc-rtems5-objcopy' : $BASE/rtems/5/bin/sparc-rtems5-objcopy
Checking for program 'sparc-rtems5-readelf' : $BASE/rtems/5/bin/sparc-rtems5-readelf
Checking for program 'sparc-rtems5-strip' : $BASE/rtems/5/bin/sparc-rtems5-strip
Checking for program 'sparc-rtems5-ranlib' : $BASE/rtems/5/bin/sparc-rtems5-ranlib
Checking for program 'rtems-ld'         : $BASE/rtems/5/bin/rtems-ld
Checking for program 'rtems-tld'        : $BASE/rtems/5/bin/rtems-tld
Checking for program 'rtems-syms'       : $BASE/rtems/5/bin/rtems-syms
Checking for program 'rtems-bin2c'      : $BASE/rtems/5/bin/rtems-bin2c
Checking for program 'tar'              : /usr/bin/tar
Checking for program 'gcc, cc'          : $BASE/rtems/5/bin/sparc-rtems5-gcc
Checking for program 'ar'               : $BASE/rtems/5/bin/sparc-rtems5-ar
Checking for program 'g++, c++'         : $BASE/rtems/5/bin/sparc-rtems5-g++
Checking for program 'ar'               : $BASE/rtems/5/bin/sparc-rtems5-ar
Checking for program 'gas, gcc'         : $BASE/rtems/5/bin/sparc-rtems5-gcc
Checking for program 'ar'               : $BASE/rtems/5/bin/sparc-rtems5-ar
Checking for c flags '-MMMD'            : yes
Checking for cxx flags '-MMMD'          : yes
Compiler version (sparc-rtems5-gcc)    : 7.5.0 20191114 (RTMS 5, RSB 5.1.0, Newlib fbaa096)

```

Fig:13 Result after configuring waf Hello World

- Now we will build application. Run the command. The output will be something close to below figure.

```
$ ./waf
```

```

Waf: Entering directory ` $BASE/app/hello/build/sparc-rtems5-erc32 '
[1/3] Compiling init.c
[2/3] Compiling hello.c
[3/3] Linking build/sparc-rtems5-erc32/hello.exe
Waf: Leaving directory ` $BASE/app/hello/build/sparc-rtems5-erc32 '
'build-sparc-rtems5-erc32' finished successfully (0.183s)

```

Fig: Result after build

- Last step is executing application with executable created after build. 'hello.exe' is the executable file name. Run below command. You will output on the console printing "Hello world". See below figure

```
$HOME/quick-start/rtems/5/bin/rtems-run --rtems-bsps=erc32-
sis build/sparc-rtems5-erc32/hello.exe
```

```

RTEMS Testing - Run, 5.1.0
Command Line: $BASE/rtems/5/bin/rtems-run --rtems-bsps=erc32-sis build/sparc-rtems5-erc32/hello.exe
Host: FreeBSD hihi 12.1-RELEASE-p2 FreeBSD 12.1-RELEASE-p2 GENERIC amd64
Python: 3.7.6 (default, Jan 30 2020, 01:18:54) [Clang 6.0.1 (tags/RELEASE_601/final 335540)]
Host: FreeBSD-12.1-RELEASE-p2-amd64-64bit-ELF (FreeBSD hihi 12.1-RELEASE-p2 FreeBSD 12.1-RELEASE-p2)

SIS - SPARC/RISCV instruction simulator 2.21, copyright Jiri Gaisler 2019
Bug-reports to jiri@gaisler.se

ERC32 emulation enabled

Loaded build/sparc-rtems5-erc32/hello.exe, entry 0x02000000

Hello World

*** FATAL ***
fatal source: 5 (RTEMS_FATAL_SOURCE_EXIT)
fatal code: 0 (0x00000000)
RTEMS version: 5.1.0
RTEMS tools: 7.5.0 20191114 (RTEMS 5, RSB 5.1.0, Newlib fbaa096)
executing thread ID: 0x08a01001
executing thread name: UI1
cpu 0 in error mode (tt = 0x101)
107883 0200b6c0: 91d02000 ta 0x0
Run time : 0:00:01.011474

```

Fig:14 Result after running executable file

We have successfully built our first application. Now we will learn how to create task, synchronization and scheduling a Task in next sections

#### 4.3 Thread Creation:

For RTEMS we make use of the POSIX APIs and hence the following guide is for the POSIX implementation. However, the documentation for classic APIs can be found on the website.

The headers `pthread.h` must be included for the following APIs.

The function `pthread_create` is used to create a new thread. It takes 4 arguments

- i. The first argument is basically a reference to a variable of type `pthread_t` and this is for identifying the created thread.
- ii. The second argument is only used if we wish to change the basic attributes such as stack size, scheduling etc. This must be passed via a variable of type `pthread_attr_t`. This can be passed as NULL and by doing so default values are chosen.
- iii. The third argument mentions the function at which the execution commences. This

indicates the start point of the thread.

- iv. The final argument is to pass an argument to the function if any.

Another useful API in conjunction with `pthread_create` is `pthread_join`. This API makes sure that the calling thread (i.e. the thread from which the API is called) is put in hold until the thread mentioned in the API finishes its execution. This API has two arguments, the first one being the thread name and the second one being an address to store the return from the thread if any. The main use of this API is to ensure that the created thread gets executed before exit or next iteration.

An example would be a main thread which runs for infinite iterations, creating new threads. In such a case it would make sense to have this API at the end to ensure that all created threads have completed before a new iteration can begin.

Note that the configuration `CONFIGURE_MAXIMUM_POSIX_THREADS` must be done prior to using threads.

#### 4.4 Semaphore creation and synchronization:

This section goes over the creation of semaphores and their use in synchronization between two threads. Note that the header `semaphore.h` needs to be included in order to use any of the following APIs. Since we also use delays for the demonstration, the header `time.h` needs to be added as well.

Before the use of any semaphore, it must be created and the API `sem_init` is used to create and initialize the semaphore with a count. It takes three arguments,

- i. The first argument is a reference to a variable of type `sem_t`. This basically acts like an identifier to the created semaphore.
- ii. The second argument is currently not used and is not described by any online documentation (Closer observation in the source files of the kernel also indicates that this argument is currently not used).
- iii. The third argument indicates the initial semaphore count.



The two basic functions used for handling semaphores are

- i. `sem_wait` which tries to lock the semaphore if it is available or puts the thread in a suspended state until the semaphore becomes available. It only takes one argument which is the semaphore identifier. Locking the semaphore is identical to reducing its count value by 1 and a semaphore can be locked if its count is more than 1.
- ii. `sem_post` is exactly the opposite to `sem_wait`, i.e. it unlocks a semaphore and makes it available for other threads to access. It also takes a single argument which is the semaphore identifier. Unlocking a semaphore increases its count value by 1.

In the demo code given below, we show how semaphores are used for synchronization. Task 1 runs continuously (periodic) and task 2 is aperiodic. For every 10 activations of task 1, we want to execute task 2. Here a semaphore is used and is unlocked by task 1 once every 10 executions. As soon as the semaphore is unlocked, the task 2 is no longer suspended. It locks the semaphore and performs its action (Which in our case is a simple print) and returns to waiting for the semaphore one again. Below is the code snippet and the result on the terminal.

```
void * task1(void * arg)

{   static int j =0;

    sleep(1);

    printf("<child>: Hello World!\n");

    j++;

    if((j%10) == 0)

        sem_post(&sem);

    return NULL;

}
```

```

void * task2(void* arg)

{   while(1)

    {

        if(sem_wait(&sem) == 0)

            printf("Semaphore received\n");

    }

    return NULL;

}

void *POSIX_Init()

{   pthread_t child1;

    pthread_t child2;

    useconds_t delay = 10000;

    int status;

    unsigned int x = 0;

    sem_init(&sem,0,0);

    status = pthread_create( &child2, NULL, task2,
NULL );

    while(1){

        status = pthread_create( &child1, NULL, task1, NULL );

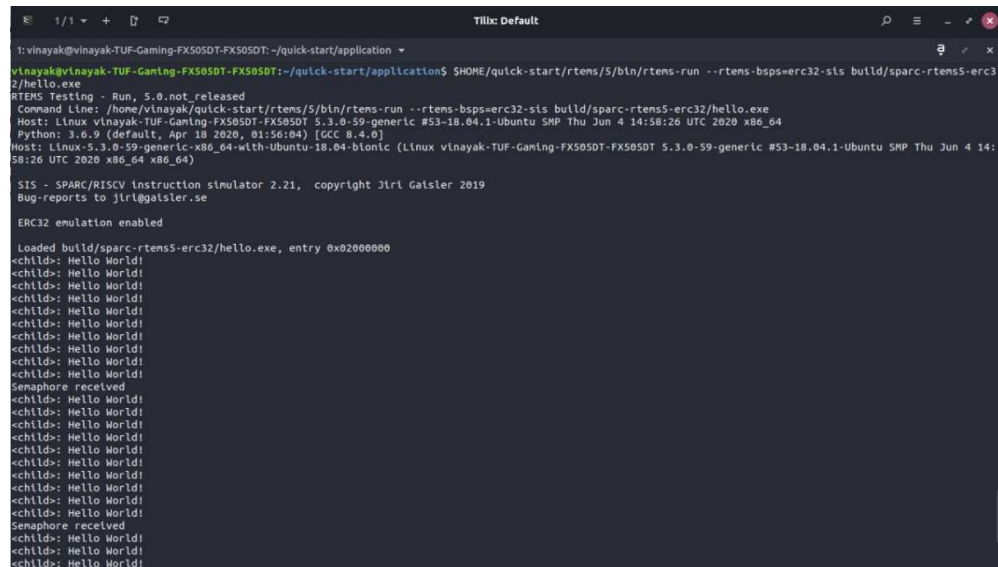
        usleep(delay);

        status = pthread_join( child1, NULL );

    }

}

```



```
1: vinayak@vinayak-TUF-Gaming-FX505DT-FX505DT:~/quick-start/application$  
vinayak@vinayak-TUF-Gaming-FX505DT-FX505DT:~/quick-start/application$ SHOME/quick-start/rtems/5/bin/rtems-run --rtems-bsp=erc32-sis build/sparc-rtems5-erc32/hello.exe  
RTEMS Testing - Run, 5.0.not_released  
Command Line: /home/vinayak/quick-start/rtems/5/bin/rtems-run --rtems-bsp=erc32-sis build/sparc-rtems5-erc32/hello.exe  
Host: Linux vinayak-TUF-Gaming-FX505DT-FX505DT 5.3.0-59-generic #53-18.04.1-Ubuntu SMP Thu Jun 4 14:58:26 UTC 2020 x86_64  
Python: 2.6.9 (default, Apr 18 2020, 01:56:04) [GCC 8.4.0]  
Host: Linux 5.3.0-59-generic-x86_64-with-Ubuntu-18.04-bionic (Linux vinayak-TUF-Gaming-FX505DT 5.3.0-59-generic #53-18.04.1-Ubuntu SMP Thu Jun 4 14:58:26 UTC 2020 x86_64 x86_64)  
SIS - SPARC/RISCv instruction simulator 2.21, copyright Jiri Gaisler 2019  
Bug-reports to jiri@gaisler.se  
ERC32 emulation enabled  
Loaded build/sparc-rtems5-erc32/hello.exe, entry 0x02000000  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
Semaphore received  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!  
Semaphore received  
<child: Hello World!  
<child: Hello World!  
<child: Hello World!
```

Fig:15 Result of threads and synchronization

Hence from the result we clearly see that for every 10 executions of the periodic thread, the aperiodic thread is being activated as the semaphore becomes available.

Note:

- The information given by the delay function on the online documentation does not match the results seen. The amount of delay generated depends strongly on the system configuration parameter “CONFIGURE\_MICROSECONDS\_PER\_TICK” and based on this setting the delay function behavior also varies. Although according to the online website the delay function used here (`usleep`) is supposed to give delay in microseconds, the actual execution does not indicate the same. Other delay functions have the same issue and requires clarification.
- Note that the configuration `CONFIGURE_MAXIMUM_POSIX_SEMAPHORES` must be done prior to using threads.

## 5. Basic installations

These are some basic tools needed to smoothly run all the above mentioned RTOS. Eclipse

is optional

- GCC on Linux. If you are not installed then you can install build essentials tools required for build using following command. This will install bunch of new packages including gcc, g++ and make.

```
$ sudo apt install build-essential.
```

- Executables are 32-bit binaries. To run a 32-bit executable file on a 64-bit multi-architecture Ubuntu system, you have to add the i386 architecture. You can get it from below command

```
$ sudo apt-get install libc6-i386
```

- Install Eclipse on Linux. Although this is not mandatory because we will mostly run the application from Terminal. Running an application from Eclipse gives you advantage of debugging also syntax mistakes can be avoided as it will be highlighted if make any of such mistakes. Use following steps to install eclipse.

- First requires a Java runtime environment (JRE) to be installed in order to run

```
$ sudo apt install default-jre
```

- Download and install the Eclipse snap package on your system.

```
$ sudo snap install --classic eclipse
```

## 6. Conclusion

The project clearly demonstrates the installation and operations of different RTOS on a linux system. This method enables users to learn more about RTOS and their scheduling mechanisms with direct hands on experience and without the need for any hardware.

However, the task of setting up the environment is quite tedious. Chances of encountering unusual errors or bugs are quite high as all the resource is free and open source and may not have the same robustness of a fully priced commercial software. A supporting online community generally exists for all this software and most forums appear to be quite active and helpful with possibilities

to discuss and solve issues with other developers alike.

## 7 References

- 1) <https://www.freertos.org/>
- 2) FreeRTOS POSIX simulator: <https://www.freertos.org/FreeRTOS-simulator-for-Linux.html>
- 3) API References for FreeRTOS <https://www.freertos.org/a00106.html>
- 4) Makefile fix for FreeRTOS <https://gist.github.com/vmayoral/32292d47b84335f698bd>
- 5) RTEMS User manual: [User Manual](#)
- 6) RTEMS installation guide: [Installation guide](#)
- 7) RTEMS detailed pre-requisite for Host-Computer: [Host computer](#)
- 8) RTEMS user manual with installation details (Quick start):  
<https://docs.rtems.org/branches/master/user/index.html>
- 9) RTEMS POSIX API: <https://docs.rtems.org/branches/master/posix-users/index.html>
- 10) RTEMS guide by Brest university <http://beru.univ-brest.fr/~singhoff/ENS/USTH/posix.pdf>.
- 11) RTEMS configuration info: <https://docs.rtems.org/branches/master/c-user/config/general.html#configure-ticks-per-timeslice>
- 12) ChibiOS User Manual: [ChibiOS User Manual](#)
- 13) How to configure ChibiOS into eclipse: <https://www.playembedded.org/blog/developing-stm32-chibistudio/>
- 14) There is forum for ChibiOS developers where you can ask doubts to Authors of ChibiOS.  
They reply you very fast. Please see all the previously asked queries. You will definitely get some hint to design your application.  
<http://www.chibios.com/forum/viewforum.php?f=13>.

15) ChibiOS forum :

<http://www.chibios.com/forum/viewforum.php?f=2&sid=6e20e8d37ef57d9648961b5e3f7a0937>

## 8 Appendix

- FreeRTOS Code:

```
/* System headers. */
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <errno.h>
#include <unistd.h>

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "croutine.h"
#include "partest.h"
#include "semphr.h"

/* Demo file headers. */

static void periodicTaskCreate(void);
static void periodicTask(void);
static void aperiodicTask(void);
//static void vTimerCallback( void );

/* Just used to count the number of times the example task
callback function is
called, and the number of times a queue send passes. */
static unsigned long uxQueueSendPassedCount = 0;
xSemaphoreHandle xSemaphore = NULL;
/*-----
*/
int main( void )
```

```

{
    xSemaphore = xSemaphoreCreateCounting(10,1);

    /* Set the scheduler running. This function will not
    return unless a task calls vTaskEndScheduler(). */
    xTaskCreate(periodicTaskCreate,"test1",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, NULL);
    //xTimerCreate("Timer1",100,pdTRUE,(void*)
0,vTimerCallback);
    xTaskCreate(aperiodicTask,"test2",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, NULL);
    vTaskStartScheduler();
    return 1;
}
static void aperiodicTask(void)
{
    while(1)
    {
        if(xSemaphoreTake(xSemaphore,(portTickType)0) ==
pdTRUE)
            printf("Aperiodic task\n");
    }
    //printf("Failed task\n");
    vTaskDelete( NULL );
}

static void periodicTaskCreate(void)
{
    portTickType xLastWakeTime;
    const portTickType xFrequency = 500;
    xLastWakeTime = xTaskGetTickCount();
    while(1)
    {

        vTaskDelayUntil( &xLastWakeTime, xFrequency );
        xTaskCreate(periodicTask,"test3",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, NULL);
    }

}
//Ignored for this demonstration
static void periodicTask(void)
{
    static int i = 0;
    i = rand()%100;
    if(i<30)
        xSemaphoreGive(xSemaphore);
    printf("Periodic = %d\n",i);
    vTaskDelete( NULL );
}

void vMainQueueSendPassed( void )
{

```

```

        /* This is just an example implementation of the "queue
send" trace hook. */
        uxQueueSendPassedCount++;
    }
    /*-----
*/

void vApplicationIdleHook( void )
{
    /* The co-routines are executed in the idle task using
the idle task hook. */
    vCoRoutineSchedule();    /* Comment this out if not using
Co-routines. */

#ifdef __GCC_POSIX__
    struct timespec xTimeToSleep, xTimeSlept;
    /* Makes the process more agreeable when using the
Posix simulator. */
    xTimeToSleep.tv_sec = 1;
    xTimeToSleep.tv_nsec = 0;
    nanosleep( &xTimeToSleep, &xTimeSlept );
#endif
}
/*-----
*/

```

- RTEMS Code:

1. **hello.c** File

```

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <semaphore.h>

sem_t sem;

void * task1(void * arg)
{
    static int j =0;
    sleep(1);
    printf("<child>: Hello World!\n");
    j++;
    if((j%10) == 0)
        sem_post(&sem);
    return NULL;
}
void * task2(void* arg)
{

```



```

        while(1)
        {
            if(sem_wait(&sem) == 0)
                printf("Semaphore received\n");
        }
        return NULL;
    }

void *POSIX_Init()
{
    pthread_t child1;
    pthread_t child2;
    useconds_t delay = 10000;
    int status;
    unsigned int x = 0;
    sem_init(&sem,0,0);
    status = pthread_create( &child2, NULL, task2,
    NULL );

    while(1){
        status = pthread_create( &child1, NULL, task1, NULL );

        usleep(delay);

        status = pthread_join( child1, NULL );
    }
}

#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER

#define CONFIGURE_MAXIMUM_POSIX_THREADS 10
#define CONFIGURE_MAXIMUM_POSIX_SEMAPHORES 1
#define CONFIGURE_POSIX_INIT_THREAD_TABLE
#define CONFIGURE_INIT
#include <rtems/confdefs.h>

```

## 2. Wscript

```

#
# Hello world Waf script
#
from __future__ import print_function

rtems_version = "5"

try:
    import rtems_waf.rtems as rtems
except:
    print('error: no rtems_waf git submodule')
    import sys

```

```

        sys.exit(1)

def init(ctx):
    rtems.init(ctx, version = rtems_version,
long_commands = True)

def bsp_configure(conf, arch_bsp):
    # Add BSP specific configuration checks
    pass

def options(opt):
    rtems.options(opt)

def configure(conf):
    rtems.configure(conf, bsp_configure = bsp_configure)

def build(bld):
    rtems.build(bld)

    bld(features = 'c cprogram',
        target = 'hello.exe',
        cflags = '-g -O2',
        source = ['hello.c',
            ])

```

- **ChibiOS Code:**

```

#include "ch.h"
#include "hal.h"
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include<unistd.h>

static void periodicTask(void);
#define SHELL_WA_SIZE    THD_WORKING_AREA_SIZE(1024)
static binary_semaphore_t sem;

    static void periodicTask(void)
{
    static int i = 0;
    i = rand()%100;
    if(i<30)
        chBSemSignalI( &sem );
    printf("Periodic = %d\n",i);
}

/*
 * periodic task thread, times are in milliseconds.

```

```

    */

static THD_WORKING_AREA(waThread1, 128);
static THD_FUNCTION(Thread1, arg) {

    (void)arg;
    chRegSetThreadName("Thread1");
    while (true) {

        periodicTask();

        chThdSleepMilliseconds(2000);
    }
}

/*
 * aperiodic task thread, times are in milliseconds.
 */

static THD_WORKING_AREA(waThread2, 128);
static THD_FUNCTION(Thread2, arg) {

    (void)arg;
    chRegSetThreadName("Thread2");
    while (true) {

        if( chBSemWait( &sem ) == MSG_OK )
        {
            printf("Aperiodic task\n");
        }

    }
}

int main(void) {
    halInit();
    chSysInit();
    chBSemObjectInit(&sem, true);
    chThdCreateStatic(waThread1, sizeof(waThread1),
NORMALPRIO+1 , Thread1, NULL);

    chThdCreateStatic(waThread2, sizeof(waThread2),
NORMALPRIO+2 , Thread2, NULL);
    while (true) {

        chThdSleepMilliseconds(5000);
    }
}

```

