Porting FreeRTOS to Portenta - Asucada

Abstract:

My end goal is to port various API's like semaphores, event flags, fixed/variable length memory pools, etc onto the Portenta Board. This would help synchronize and communicate effectively between layers and devices on task systems. This has already been done for ARM Cortex-M4F and STM32F7 ARM Cortex-M7 based microcontrollers. I will use this as a starting point and implement the same on the Arduino IDE.

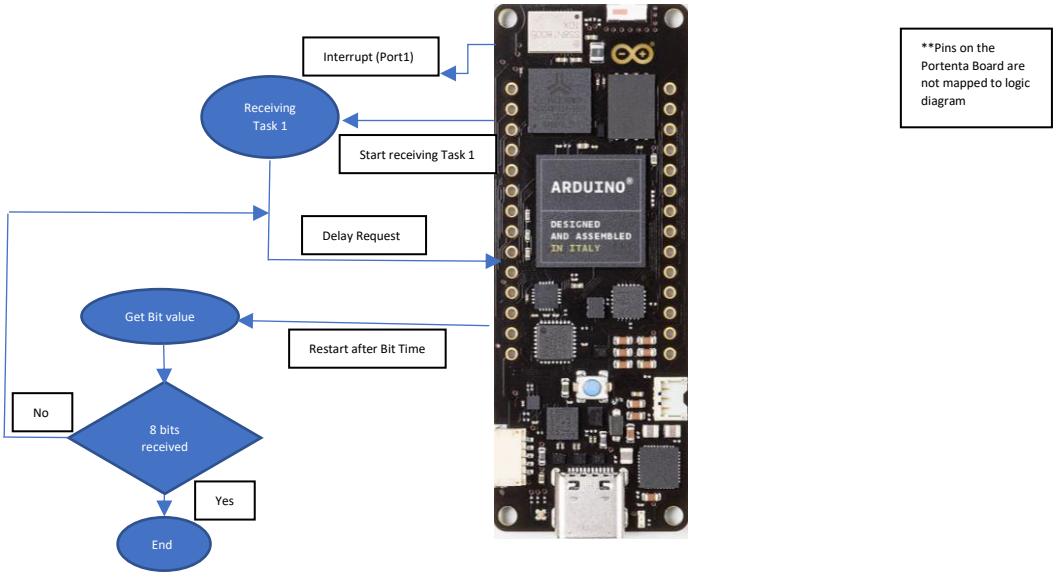
Technical Details:

Introduction

The key feature of an RTOS is to break down the application into several tasks or, different threads of execution. This is much more maintainable as one is relying on the services of the kernel to do all the signaling and timing. The diagram is a depiction of a task being executed. Once a system call is made, the following steps occur:

1. Interrupt set high on Port 1
2. First bit received
3. Delay for one clock cycle
4. Repeat until 8 bits received

If a second task is initiated while Task 1 is executing, bit transfer can still occur during delay time of Task 1. This example captures the reusability and reliability of an RTOS.



A typical RTOS comprises of file systems, command line interface, TCP/IP. However, FreeRTOS is primarily based on a scheduler and a group of services around that scheduler.

FreeRTOS takes 70 bits of RAM per task. It additionally has 5-9.5K of flash.

The Portenta Board is an ideal candidate for FreeRTOS due its following specs: 1.STM32H747 dual-core processor with graphic engine, 2.8MB SDRAM, 3.2-128 Mbyte QSPI flash

Work in Terms of FreeRTOS

The files that are needed to be written to port FreeRTOS to Portenta for core kernel functionality include:

1. Source Files
    a. ISR (Interrupt Service Routine)
    b. Queue
    c. Tasks
    d. Timers(optional)
2. Port File
3. Configuration File
4. Memory Management Function

Given below are details of the above files.

Interrupt Vectors

Interrupts are an essential part for the RTOS to work. A timer is used to generate a periodic tick interrupt and, additional interrupts are used to manage context switching. These interrupts are serviced by the RTOS port source files.

Additionally, Interrupt Nesting Model needs are created to set, 1. How you want the nesting model to work as they are a range of interrupts that are never disabled, 2. Set the interrupt priority for every interrupt you are using to set it in the correct range

Configuration File

FreeRTOS is customized using a configuration file called FreeRTOSConfig.h. Every FreeRTOS application must have a FreeRTOSConfig.h header file in its pre-processor include path. FreeRTOSConfig.h tailors the RTOS kernel to the application being built. It is therefore specific to the application, not the RTOS, and should be in an application directory, not in one of the RTOS kernel source code directories. Some examples of these files include Memory allocated related definitions, Hook function related definitions, Run time and tasks stats gathering related definitions, etc.

Memory Management Function

Different applications need different memory management functions or different RAM allocated to the kernel. This is because several applications will create data in the beginning and never create

anything again. Some applications have memory coalescence where if you create and delete memory very quickly, it'll try to match up adjacent blocks, so you don't get memory fragmentation.

FreeRTOS already includes many demo applications targeted at: 1. Microcontroller, 2. Development Tool, 3.Hardware Platform. It is a simpler task to take an existing demo for one evaluation board and modify it run another. Here is a link to the supported devices: https://www.freertos.org/a00090.html. Out of these ARM Cortex-M4F and STM32F7 ARM Cortex-M7 based microcontrollers support dual-core. The steps involved in this transfer include:

1. Initial compilation of existing demo application
2. Modifying the LED IO: easiest way to get visual feedback that the demo application is running by changing pin configuration, requires editing the functions vParTestInitalise() within parset.c and prvSetupHardware() within main.c
3. Introducing the RTOS Scheduler: flash test task that toggles a LED with mixed frequency by calling function vStartLEDFlashTasks()
4. Restore the full demo application with new demo tasks created

In case this doesn't work out, I can also create a new FreeRTOS Port by referring here: https://www.freertos.org/FreeRTOS-porting-guide.html

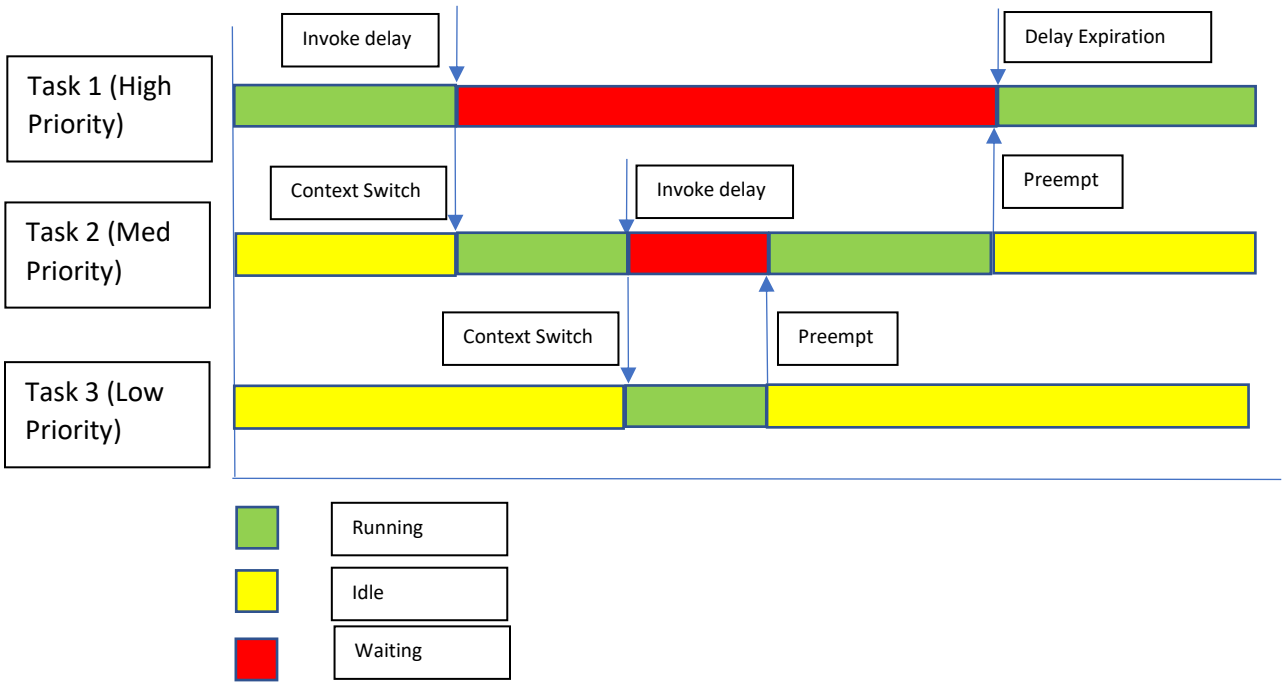
Work in Terms of RTOS

In order to test these files, various tasks need to be defined. This is done through a Task Control Block (TCB) which has the following structure:

Pointer to previous TCB
Pointer to next TCB
Task ID
State
Priority
Program Address at start of Task
Stack Pointer at start of Task
Control Regulator at start of Task

Task States are typically of 4 kinds: Ready, Running, Waiting and Dormant. This tells us whether to transfer information from the local memory to the CPU or not.

Scheduling plays a huge role on the order of which tasks are done. Below is a diagrammatic representation of context switch, a method to manually move between tasks with different priority.



The order of priority of task is as follows: 1.ISR(Interrupt Service Routine), 2.Handler, 3.Task. In case multiple tasks have the same priority, a queue needs to be created which follows FCFS(First Come First Serve). Types of handler include: 1.Interrupt Handler, 2.Cyclic Handler, 3. Exception Handler.

Like I have mentioned in my abstract, my final goal is to port various API onto the Portenta. Some of these include:

1.Activate: moves the state of the task from dormant to ready

2.Self Terminate: moves the state of the task from running to dormant

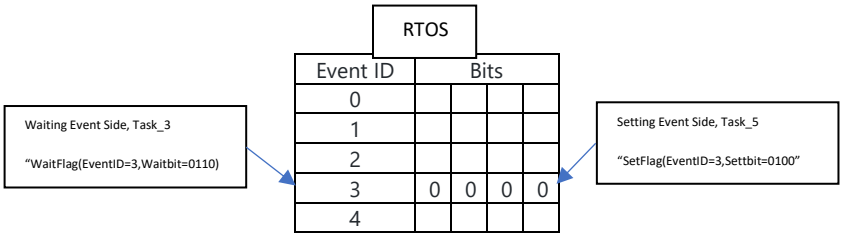
3.Semaphore: provides exception control in the shared memory by giving semaphore to the requesting task

When a task wants to acquire a semaphore, it does this with an ACQUIRE SEMAPHORE API. If a semaphore is available, the RTOS returns succeed which corresponds to giving a semaphore. If the semaphore is unavailable, that task ID is put to waiting. In case more than one task is waiting for that semaphore, preference is given to the highest priority task.

Semaphores can be released by task that didn't originally acquire it or, can also be released by an ISR. This is a key example of how to make software architecture better and software more reliable. By the correct orchestration of semaphore, task performance is significantly improved.

4.Time Flag: raises a flag for signaling other tasks

This is controlled by 2 different APIs, WaitFlag and SetFlag. Below is an example of how it works.



The steps involved in this are:

1. Task 3 wants to run a process but before this Task 5 must finish. Hence, Task 3 is in the waiting state
2. SetFlag(EventID=3, Setbit=0100) is called and 0100 is written in Event ID 3
3. Task 3 is unaffected as this doesn't match pattern for WaitFlag(EventID=3, Waitbit=0110) which is 0110
4. SetFlag(EventID=3, Setbit=0010) is called and 0110 is written in Event ID 3
5. Task 3 is released from waiting and put into running

A key takeaway is that an ISR can invoke the API. Also, RTOS must check for tasks that are waiting for different flag patterns in the same queue. Hence, this is not simple FIFO.

5.Mailbox_Exchange: send messages between tasks

6.Fixed/Variable Length Memory Pool: Used to dynamically assign fixed or variable size memory areas to each task

7.Sleep/Wakeup

8.ChangePriority

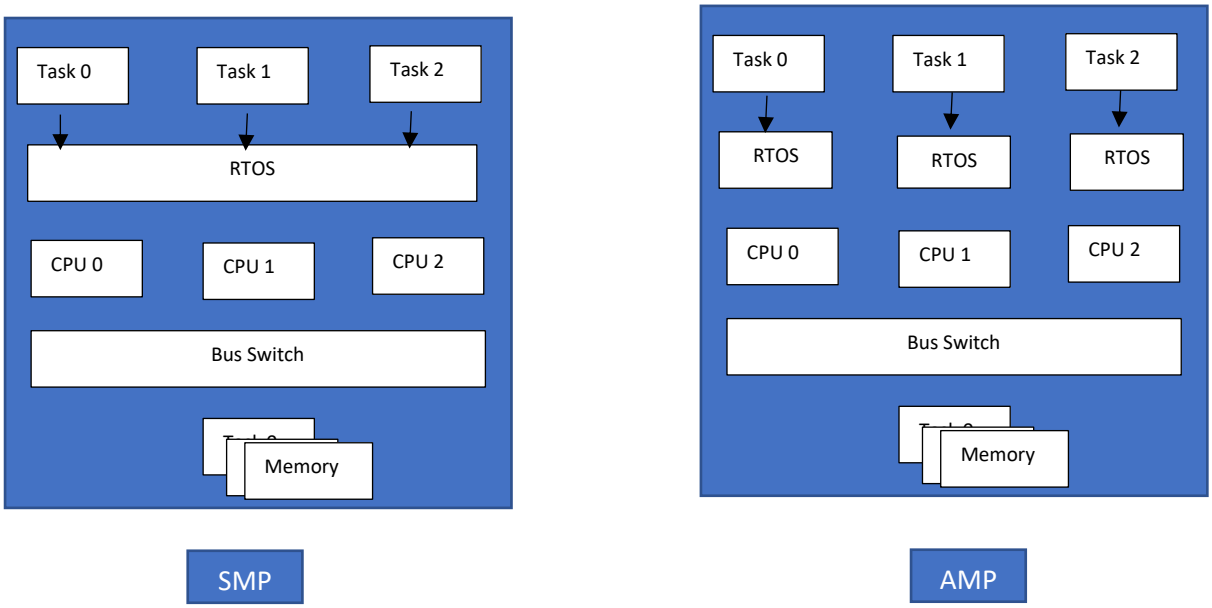
9.RotateReadyQueue

10.LockCPU

An important functionality which branches into all the other APIs are TimeOut and Polling. TimeOut moves the task from waiting to ready state after a set time. Even if a software lag or other exception occurs and prevents the wait condition from changing, a lock can be avoided. Similarly, polling uses features of API that transition to Waiting even if ISR or Cyclic Handler are evoked.

Since the Portenta has a dual core, I can also enable a multicore ready RTOS. There are 2 types that can be built: SMP (Symmetrical) and AMP(Asymmetrical).

The structures of both of these are as follows:

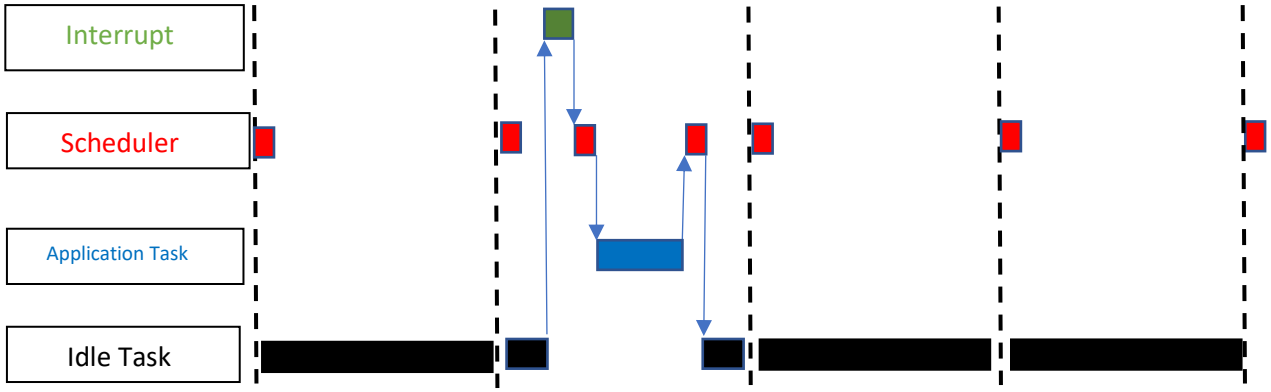


In SMP, the task is assigned to any available CPU. Any task can be run by any CPU as the load balance is handled dynamically. There is low real time performance if the execution code and date aren't in the cache. In order to solve this, one has to execute cache coherence.

On the other hand, in the AMP, software that'll run on each CPU is already determined as the RTOS differs for each CPU. If for example, CPU 1 has a heavy load and CPU 3 has a light load, the task can change between them to process faster. Hence, this has high real time performance.

Miscellaneous

An additional feature I'd like to add is a Tick Scheduler for low power applications. This works by setting the board to a low power state when the board is idle or not pursuing a task.



Goals once these tasks are complete:

If time prevails, I would like to port RTOS+ onto the Portenta board. Some of the features that would be added include:

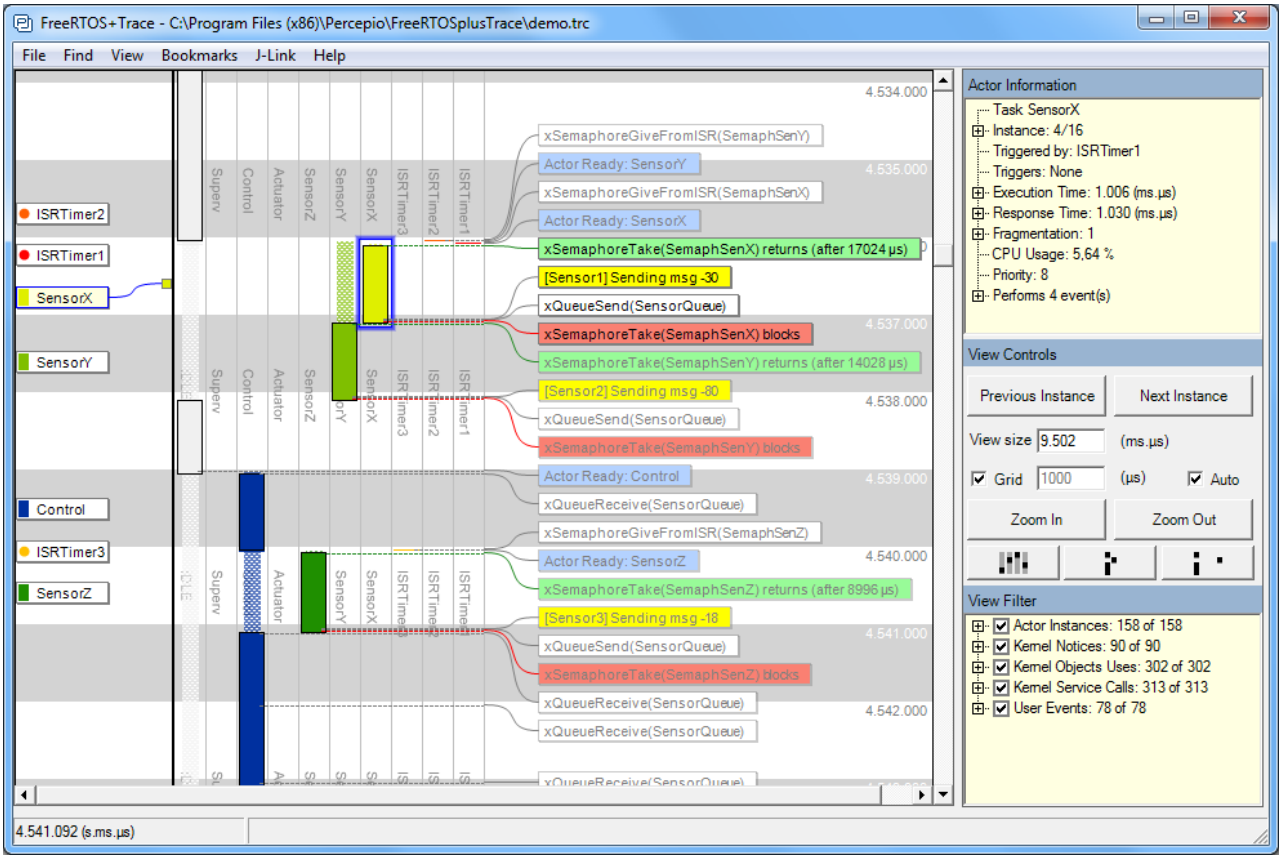
1.TCP/IP: Pre-built libraries for easy integration of TCP/IP(Transmission Control Protocol/Internet Protocol) into cost-sensitive applications

Internet traffic needs to be routed in the right way. Top layer called application which programs like web browser directly interact with. Has protocols has HTTP/SMTP. Next layer is called Transport where the TCP/UDP live. Used for low latency applications like online games. Divides it into packets so they can be sent. To put back together, TCP slaps a header.

(Portenta Board supports Wifi at 65Mbbs transfer rate)

2.CLI: Enable your application to efficiently process command line input

3.Trace Tool: Optimization, Debugging



Schedule of Deliverables:

Time	Work
April 27 – May 18	-Community Bonding Period -Purchase ARM Cortex-M4F or STM32F7 ARM Cortex-M7 based microcontrollers and start testing FreeRTOS to get familiar with the environment -Start editing port files according to pin configuration for Portenta Board available online
May 18- May 25	- Modifying the LED IO
May 26- June 1	- Introducing the RTOS Scheduler
June 2-June 8	-Final edits to source file, configuration file and, memory management function
June 9-June 15	Prepare for Phase 1 Submission
June 16- June 22	-Introduce API Activate, Self-Terminate and Semaphore
June 23 -June 29	-Introduce API Time Flag and Fixed/Variable Length Memory Pool
June 30-July 13	-Test created APIs with task defined on the TCB
July 14- July 17	Prepare for Phase 2 Submission
July 18-July 25	-Enable SMP (Symmetrical multi core processing)
July 25 – August 1	-Enable AMP (Asymmetrical multi core processing)
August 1 – August 8	-Introduce Tick Scheduler for low power applications
August 8-August 15	-Integrate Trace Tool -Prepare for Final Submission

Development Experience:

I have experience in both high and low levels of programming.

Last summer, I was working on an educational kit for embedded programming called QUARKit. I had to develop firmware for the Atmel AT89S51/C52 and various other components like 7 segment display, bar graph monitor, LCD Screen, EEPROM and Keypad Interface. I developed exercises in order to learn how to interact with these different I/O devices. Later, I went to deliver a workshop in one of India's most prestigious universities, IIT ISM Dhanbad. The best part about embedded programming is that you can see your hardware come to life when you burn your firmware onto the microcontroller. This provides a physical sense of feedback.

In my sophomore year, I took the course Microprocessors and Interfacing course. As a part of this, I had to complete a project based on low level programming. My task was to implement a door security system using software Proteus 8. In order to do this, I had to develop code in assembly. You can check this out on my Github: https://github.com/asucada

I have also used both Arduino Uno and Mega for my projects. I used to be part of a technical team called Team BITS at Shell Eco Marathon. We have developed India's first all-ethanol car. My job was to use Arduino boards to:

1.Create a lap counter with real world time on an LCD display

2.Interface various sensors to detect overheating and position of car

I love how Arduino is very easy to get started with. It opens up the world of embedded programming with a fun approach.

Why this project?

This project falls in the goldilocks zone for me. I know enough about it to get started and I can learn so much more by doing it. I got a real sense of communicating between hardware by working on the QUARKit. I want to go one layer deeper and work on operating systems. This is the perfect opportunity for me to take my programming skills to the next level.

When I was working on the QUARKit, an embedded design educational kit, I also got a real sense of documentation. Its very important that your work is transferable to others. I had to explain each line of code I wrote in the QUARKit manual. I believe this will help me develop well-documented code for Arduino.

Can't wait to get started!

Other Commitments

I'm one of the 50 people selected globally to carry out a research project under the mentorship of Dr. Stephen Wolfram, Founder of Wolfram Research and Creator of Mathematica. This is a 3-week long program held in Bentley University in Waltham, MA from June 28 – July 17, 2020.

To make up for the time lost in this program, I will work extra hours during the rest of the summer. I'll plan my time accordingly so I can take my GSoc project to completion.