UNIVERSITY OF MINNESOTA
Driven to Discover℠

FreeRTOS: Installation and Implementation on Pololu A-Star 32U4

John Undersander
CSCI 5143
Spring 2018

Motivation

This project utilizes a Real-time Operating System (RTOS) which provides a number of strategies covered throughout Real-time and Embedded Systems such as static and dynamic task scheduling, preemptive kernel behavior, and critical section management [5]. FreeRTOS is an open source project provided under the MIT license, and is used in industry by many companies and organizations to create numerous embedded systems solutions. After learning about the process of task creation and building our own static scheduler into the course projects this spring, the notion of using an RTOS on our board intrigued me, as it provides an API for accomplishing the same goals with faster development and greater deterministic behavior out of the box. Recent experiences in other courses have piqued my interest in systems programming and kernels, and with FreeRTOS being a relatively small open source kernel, it presents itself as an appropriate depth to explore and learn. Given that this RTOS is currently the de facto market standard [4], there is also a good chance that I will see it in my career at some point; therefore, getting familiar with how it works and what it can and cannot accomplish is a valuable goal in and of itself.

Outcomes

Makefile

This kernel and port are written entirely in C and therefore require use of a Makefile to practically build the final product across multiple environments. Before this work, I had elementary understanding of the process of Makefiles from other courses. These courses often emphasised other programming concepts without much focus on specifics starting from scratch; therefore, I was usually provided a working Makefile and told to simply use the typical commands for each target (make all, make clean, etc.). I started this project by first searching YouTube for a quick refresher tutorial on Makefiles and found one by Barry Brown [2] which very succinctly introduces the topic. More importantly, he describes not only how, but *why* things are done this way and in what order by a typical C compiler/linker.

FreeRTOS recommends that a new project be developed from one of their pre-existing demo applications if possible. A quick search revealed an existing port for a board that uses the ATMega323 microcontroller [1] which happens to share a similar pinout as our ATMega32U4. I utilized this by comparing the demo Makefile with the Makefile our class had been using for our projects all spring. I simply needed to include a lot more than just main.c as our labs had been doing. Through this process, I learned various strategies such as how to specify substitutions [7], which can significantly shorten target definitions in a given file. The example Makefile grouped required files and directories together by specifying path root variables (ex. FreeRTOS Source directory) and then listing the required files as relative paths to that variable name. This is important, as it allows a developer to change the project directory structure in the future, hopefully only requiring one variable change if the Makefile was designed with this in mind. I enjoyed this process, as it required reading another developer's work and translating it into the problem domain I was working in. I feel that this is a skill I will be utilizing frequently in industry.

Configuration

The configuration step in the port process was expected to be the most time consuming by far, as it requires interfacing the FreeRTOS kernel with a specific hardware chip. The core of the kernel is comprised of only three C files, requiring inclusion of one of five provided heap implementations, and a working port only requiring one hardware specific file, port.c, as per the documentation [3]. An application specific config file, FreeRTOSConfig.h, is also required. Before starting on these two required files from scratch, I searched the Internet for anyone who has endeavored the same task in the past in order to not reinvent the proverbial wheel. The first result from Google was an AVRFreaks thread with a user inquiring about the port.c file specific for ATMega32U4 [10]. Luckily for this user and I, as stated earlier, the demo application for the ATMega323 provides a very close match to the required port.c file for our chip. Using this and the information from the FreeRTOS documentation, I was able to get pointed in the right direction with this port much faster than originally anticipated. The documentation notes regarding interrupt vectors pointed to the prvSetupTimerInterrupt method in the port.c file. This is specific to each chip and is used to setup a timer used for generating tick interrupts in the system, much like our scheduler was doing in our labs and projects. Coincidentally, this port was already setup for Timer 1 using compare match A, which is the setup I had used for my scheduler throughout the course, so I easily recognized the named bits and vectors that needed to be changed.

The AVR forum thread saved quite a bit of debugging time during the port process. I had been trying to get a simple led task to run with the macro portCOMPARE_MATCH_A_INTERRUPT_ENABLE set to OCIE1A, our board specific method to enable interrupts on timer 1 mask register (TIMSK1) which I used several times in projects. Pit_21 from AVRFreaks used a simple bit shift (1<<1) for the definition of this macro without using the avr/io named bit macros, so it was not obvious at first to look closely at this. With the lower bits of the timer mask set to 3 instead of 2, the timer overflow bit is set (TOIE1) which enables use of the overflow vector by developers. I was surprised that this one bit being enabled was the difference between the scheduler running and not running, and I later determined that it was due to inclusion of the FreeRTOS timers.c functionality in the project. For some reason, the kernel relies on the overflow interrupts being enabled when this file is included. I am assuming this is done to force the application developer to handle overflows on their own timers, but further research would be needed to conclude. FreeRTOS itself does not use the overflow ISR method. Instead, the kernel is examining the tick count at each timer interrupt, which allows it to handle an overflow event inside of the tick incrementing routine when it occurs. Going forward, I will be more skeptical of previous working configurations when using a third party API. Moreover, I will ensure to fully examine the methods of others when they conveniently share their work as a means of help so that I do not miss any important specifics. Following this process, all it took was looking closely at compiler errors and subsequently modifying the FreeRTOSConfig.h file to fix them based on my setup. I have commented this file with all of my changes that allowed the port to minimally function with the RTOS scheduler.

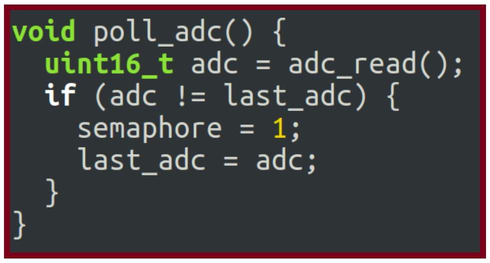
Context Switching

Before I started looking into RTOS code at all, I had wondered how one would implement a preemptive scheduler on a single core chip. We had spoke about these topics at a high level in class and had even built our own non-preemptive schedulers on our boards, but looking at the C code I had written, it seemed impossible to do in a similar manner. It wasn't until I began digging through the port.c file for the AVR and stumbled upon some Assembly code blocks dealing with saving and restoring context [11] that a revelation was achieved. It is quite obvious now that an action as intrusive as stopping what the processor is currently doing and starting something new must be done with processor level instructions. It has been a while since I have written Assembly back in Machine Architecture class, but due to the great FreeRTOS documentation and relatively small amount of Assembly required for a port, I am confident that I could translate the examples onto a different board if I ever came across such a task.

I find it fascinating how the kernel can push all of the registers, including the status and current program counter, onto each task's corresponding stack. FreeRTOS then maintains a Task Control Block (TCB) structure for each task, allowing it to switch between the contexts of each task and restore the correct location in memory to resume execution. Learning about these strategies has given me a new perspective on timing in a system and how fast modern machines are getting. The work required to save, switch, and restore context looks like a huge overhead on paper; however, our little 16Mhz chip is able to maintain multiple simple tasks (such as LEDs) executing within milliseconds of each other without major system stability changes. In fact, it seems that the biggest limiting factor of our chip is RAM [6], with FreeRTOS only being able to manage separate stacks for a little over 10 tasks on such a system (2.5KB RAM) depending on the tasks. I can then imagine our smartphones, with multi-core 2+Ghz chips and multi gigabytes of RAM, and the power they have to execute tasks over our boards. I can remember individuals telling me in Machine Architecture that I should not worry about the course content, as I will not be seeing it much after the class. I disagree, as I think it is important for a Computer Scientist to keep a close relationship with the hardware they work on and understand the limits it imposes.

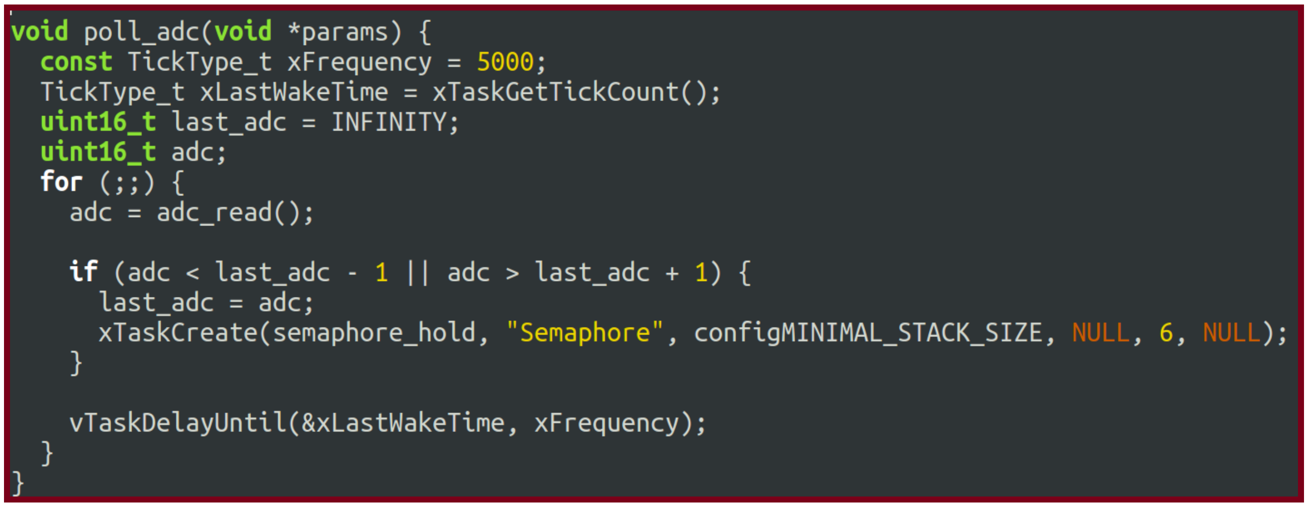
Product

Once the RTOS was functioning on the A-star, my goal was to implement the requirements (or at least a subset) of project 2 [9]. The tasks were already written and just needed to be translated to follow the FreeRTOS contract. Each task needs to be implemented as an endless loop and must not return [8], as the scheduler is designed to always be running a task and does not know how to deal with returning tasks. A positive side effect of this requirement is that it allows encapsulation of all required global variables into the function scope of the task that require them. An example of this is the last_adc variable used in the ADC Poll task below (Figure 1 vs. 2). To implement the periodic nature of our project, the API



```c
void poll_adc() {
  uint16_t adc = adc_read();
  if (adc != last_adc) {
    semaphore = 1;
    last_adc = adc;
  }
}
```

Figure 1: Implementation of ADC task in project 2



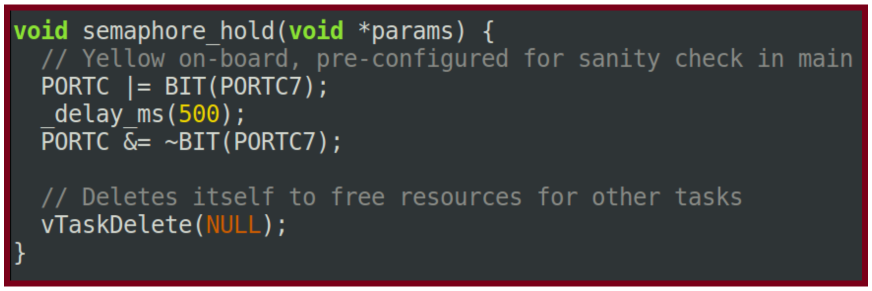
```c
void poll_adc(void *params) {
  const TickType_t xFrequency = 5000;
  TickType_t xLastWakeTime = xTaskGetTickCount();
  uint16_t last_adc = INFINITY;
  uint16_t adc;
  for (;;) {
    adc = adc_read();

    if (adc < last_adc - 1 || adc > last_adc + 1) {
      last_adc = adc;
      xTaskCreate(semaphore_hold, "Semaphore", configMINIMAL_STACK_SIZE, NULL, 6, NULL);
    }

    vTaskDelayUntil(&xLastWakeTime, xFrequency);
  }
}
```

Figure 2: Implementation of ADC task using FreeRTOS

provides a method, vTaskDelayUntil (Figure 2), which specifies a time in the future that the task will unblock. While the task is delayed, it will not be scheduled and will receive no CPU time. It is much simpler to add the task to the scheduler with FreeRTOS as well. A simple call to xTaskCreate specifying the required parameters and then the scheduler can be started. This function [12] takes six parameters [13], one of which is the tasks stack size in words. I left this value to be the value recommended by the FreeRTOSConfig.h file, figuring that it could easily be increased per task if needed. In a real world case, one would want to use the tools provided by FreeRTOS to tune the stack size for each task in order to squeeze the most out of the limited resources.



```
void semaphore_hold(void *params) {
  // Yellow on-board, pre-configured for sanity check in main
  PORTC |= BIT(PORTC7);
  _delay_ms(500);
  PORTC &= ~BIT(PORTC7);

  // Deletes itself to free resources for other tasks
  vTaskDelete(NULL);
}
```

Figure 3: Implementation of semaphore task using FreeRTOS

My implementation of the semaphore task from project 2 (Figure 3) is one that gets created upon the ADC poll task sensing a change. In this way, I was able to utilize the heap_4 memory implementation [3], which allows for dynamic RTOS heap with a coalescence algorithm. The semaphore task may not need to run indefinitely, so it can be deleted (in fact it deletes itself) to free resources for other tasks. This was a fun exercise, but in a typical embedded application with constrained resources, dynamic memory manipulation is not welcome. An alternative, more deterministic approach would be to utilize a similar pattern with vTaskSuspend and vTaskResume [5] in place of creation and deletion. This way, it allows the system to contain all of the memory needed for the semaphore task statically allocated, and simply give the task no processor time until it is needed. The task could then suspend itself, essentially going dormant until the next change of the ADC reading.

Future Work

This project has most certainly sparked an interest in RTOS going forward. What I have produced is a good start to a base project from which I can fork and try out any number of variations of features that the FreeRTOS ecosystem offers. One specific feature, which I did not get a chance to implement here, is the run time statistics that are available as task utilities [5]. Much like we did in project 2, I could analyze how the system is behaving beyond just observing the blink rate of LEDs. RTOS are used in many space related applications such as ground based rovers and UAVs. Space has increasingly been an industry of interest in my life, and while I may not be able to replicate the Mars Curiosity rover with the Pololu A-star, this board could serve as the brain of a satisfying (yet small) mission critical project to showcase my interest to friends, family, colleagues, or even future employers. The end of RTES, and soon my college career, only marks the start of my life long learning journey.

References

[1] Atmel AVR (MegaAVR) / WinAVR Port [RTOS Ports]. (n.d.). Retrieved May 7, 2018, from https://www.freertos.org/a00098.html

[2] Brown, B. (2016, April 03). C Programming: Makefiles. Retrieved May 7, 2018 from https://youtu.be/GExnnTaBELk

[3] Creating a New FreeRTOS Project [More Advanced]. (n.d.). Retrieved May 7, 2018, from https://www.freertos.org/Creating-a-new-FreeRTOS-project.html

[4] Embedded Staff. (2017, July/August). 2017 Embedded Market Survey. Retrieved May 7, 2018, from https://www.embedded.com/electronics-blogs/embedded-market-surveys/4458724/2017-Embedded-Market-Survey

[5] FreeRTOS API [API Reference]. (n.d.). Retrieved May 7, 2018, from https://www.freertos.org/a00106.html

[6] FreeRTOS FAQ relating to FreeRTOS memory management and usage. (n.d.). Retrieved May 8, 2018, from https://www.freertos.org/FAQMem.html#RAMUse

[7] GNU make [Substitution-Refs]. (n.d.). Retrieved May 7, 2018, from https://www.gnu.org/software/make/manual/make.html#Substitution-Refs

[8] Implementing a Task [More about tasks...]. (n.d.). Retrieved May 8, 2018, from https://www.freertos.org/implementing-a-FreeRTOS-task.html

[9] Larson, A. (2018, March 6). RTES Scheduling Lab: Counters, Timers, and Scheduling. Retrieved May 8, 2018, from https://github.umn.edu/umn-csci-5143-S18/public-class-repo/blob/master/project2/README.md

[10] pit_21, & clawson. (2014, August 28). FreeRTOS Atmega32u4 project setup. Retrieved May 7, 2018, from https://www.avrfreaks.net/forum/freertos-atmega32u4-project-setup

[11] Saving the RTOS Task Context [RTOS Implementation Building Blocks]. (n.d.). Retrieved May 8, 2018, from https://www.freertos.org/implementation/a00016.html

[12] Undersander, J. (2018, April 30). Final_project/main.c. Retrieved May 8, 2018, from https://github.umn.edu/umn-csci-5143-S18/repo-under214/blob/1dc7b21b359f1c05522278f729488e4e36c4b33d/final_project/main.c#L76

[13] xTaskCreate [Task Creation]. (n.d.). Retrieved May 8, 2018, from https://www.freertos.org/a00125.html