CSCI 5991, 2.0 Credits - The Zephyr RTOS

Semester: Spring 2025
Supervising Professor: Jack Kolb
Student: Adam Arnett

Over the course of the semester, I have studied the Zephyr Real-Time Operating System through the exploration of examples and creation of new drivers. Initially I intended to look at the RTOS as a whole and examine many different parts of it, however I ended up focusing primarily on drivers and devicetree. I'm not unhappy with this result, because devicetree is notorious for having a difficult learning curve which I think I got over. My performance during this semester didn't meet my initial expectations, but my expectations were too high to begin with. I have a history of being too optimistic in terms of how much programming I can do in a given amount of time, and this independent study has helped me better understand this. I also repeatedly ran into situations where I couldn't really proceed further with what I was doing due to some sort of technical limitation.

In order to run code, I used the nRF52840-DK by Nordic Semiconductor. I chose this board because much of the sample code within the official Zephyr repository is supported on that development kit. I also used several peripherals (including but not limited to the MightyOhm Geiger Counter, Sensirion SGP30 gas sensor, Orient Display AC780S driven LCD displays, and Microchip MCP9808 thermometer) which I connected to the nRF52840 via jumper wires and a breadboard.

Before getting to work, I had to set up the development environment, which wasn't exactly easy. There are two main options, either using Zephyr's West tool on the command line through a python virtual environment, or using Nordic Semiconductor's nRF connect extension in VS code. Initially I went with the former, because I like using the command line a lot, but in order to be able to right click and jump to a definition of something in the Zephyr SDK I had to switch to nRF connect. This also had the bonus of a visual representation of devicetree nodes (things like specific GPIO ports, I2C controllers, etc) which made learning about devicetree much easier. I will cover devicetree more later. Both of the two options I mentioned were relatively complicated to get set up, but I think the command line approach was a little easier since it didn't require the installation of proprietary software. In order to get the nRF software working, I had to read some error logs since it didn't work as advertised and then I had to manually change the value of something in some random file within /etc on my computer. I think it was the version of a package or library.

During the first half of the semester I mostly focused on learning about Zephyr in general. Many of the operating system concepts are extremely similar to things I saw in Linux in CSCI 4061. The main differences lie in a lack of a fully featured virtual memory, and a simpler userspace/permissions system. Things more related to embedded systems like I2C and GPIO were quite different compared to what I saw in CSCI 5143.

In terms of virtual memory, Zephyr does support it, but only on systems that have a memory management unit. If this unit is present, Zephyr can map virtual memory, but only in a 1:1 ratio between virtual and physical memory. This limitation can be bypassed by enabling demand paging, but I never dug too far into that. In terms of userspace, Zephyr has a user mode and kernel mode. Threads can be run in either, but if a thread is started in user mode, it does not have access to kernel objects such as semaphores, timers, etc by default. Access can be granted by a kernel thread calling a function to give access to a specific instance of an object, or a user thread can be upgraded to a kernel thread which grants access to all kernel objects. By default there is no filesystem in Zephyr, so no read/write/execute permissions. Zephyr can mount FAT or Ext2 file systems, but in general any thread could technically access any memory if a memory management unit was not being used.

Getting things like GPIO and communication protocols like I2C were quite difficult at first, due to the fact that Zephyr uses devicetree to organize hardware description/access. This allows Zephyr to have generic APIs for everything from GPIO to LTE modems. This is advantageous because it allows peripherals to be swapped out without necessitating changes to code - just the devicetree files instead. So if I was prototyping something and suddenly realized a sensor I was using was no longer appropriate for the project, so long as a driver exists for the new sensor I'm using I could change as little as one line (the compatibility option in the devicetree overlay) and the new sensor would be working in the same way the old one was. This can save quite a bit of time, but not everything has a driver ready to go, and using devicetree adds several steps to something that could be quite simple such as GPIO. I wouldn't say it was difficult to get GPIO running (responding to button presses, blinking LEDs, etc), but it took much longer because it had to go through devicetree. Getting GPIO to run for the first time via bare metal programming on the AVR-BLE (CSCI 5143) probably took about three minutes, but it was probably closer to three hours with Zephyr on the nRF52840. There are many different devicetree files a programmer needs to think about, but the primary one is called an overlay. Other devicetree files will describe a board and its default hardware/setup, but the overlay file basically overrides any defaults specified in other files found in the zephyr SDK. For example, the default pins for the I2C1 peripheral on the nRF52840 are P0.31 for SCL and P0.30 for SDA. If I'd rather use P0.27 and P0.26, I can redefine I2C1 in my overlay file to use those pins instead. I could also change or add other properties like the speed or number of addressing bits. No matter what kind of peripheral I'm using and through which pins I'm accessing it however, an overlay file needs to be given in the vast majority of cases, since many peripherals will have the status property set to "disabled", which must be overridden with "okay" in order for it to be included in the build. Similarly and for the same reason, peripherals must be enabled in kconfig configurations. If I want to use an I2C peripheral, I will need "CONFIG_I2C=y" in a separate file in order for I2C to work.

During the second half of the semester, I focused on making a driver that I could contribute to the official Zephyr git repository. Originally I intended to create one, upstream it, and focus on something else like power optimization, but I ended up creating four drivers total

instead and upstreaming none of them because it turned out they already existed. The first was for a 14-segment display made by Sparkfun. This didn't work out, since all the LEDs in the display were controlled by an IC called the VK16K33, which is compatible with another IC called the HT16K33, which already had a driver in the repo. It existed in a place I didn't really know to look in. I tried again with an LCD display. I finished the driver and just needed to test it, but someone on the Zephyr Discord channel pointed out that it may be compatible with a driver that already existed. I didn't think it was since the datasheet never mentioned that it was, but I gave it a shot and it was compatible with a much more common protocol. By this point I had gotten much more comfortable with drivers, and could get one running to a minimal and buggy degree in a matter of hours instead of days. The last two drivers I made were for the SGP30 gas sensor and MCP9808 thermometer. Originally the SGP30 was just going to make a thermometer more complicated so I could work on analyzing power consumption. It's an obsolete part so I wouldn't expect a driver for it to be accepted into the Zephyr repo. By chance, I came across the MCP9808 in my drawer at home. I think I had used it for an Arduino project years ago. I found that it was not obsolete, did not have a driver in the repo yet, and was not compatible with any extant drivers. I immediately started working on a driver for it since I really wanted to get something contributed. Unfortunately I ran out of time. I still plan on trying to finish the driver for the MCP9808 and get it contributed.

While working on drivers in the second half of the semester, in addition to learning about drivers themselves, I learned a lot about devicetree, kconfig, how the Zephyr APIs work, and got much more comfortable with a bunch of preprocessor macros related to how devicetree nodes are instantiated. I'm really proud of myself here, since at the beginning of the semester I didn't even know how to start writing a driver, and now I can write one with ease! Each C file that contains the source code for a driver basically contains implementations of generic API functions (sensor_sample_fetch for example), helper functions for those functions, and the use of a macro that instantiates instances of the driver based on devicetree nodes compatible with the driver. That macro totally confused me and I basically just changed variable names from a sample at first in order to get it to work. In retrospect, I understand why I thought it was difficult at first (I think macro definitions that are 20 lines long and use token pasting would make most engineers sweat) but all it actually did was create a data and config struct in addition to creating an instance of the specific driver through the API and devicetree.

Part of the inspiration for this independent study was because the company I work for is considering switching from bare metal programming to using an RTOS (possibly Zephyr) in their products in the future. Now, I ask myself, "Is Zephyr appropriate?" The answer is no for now. I work for a medical device company and since Zephyr is not yet certified for use in medical devices, the answer is an easy no. If it was, I think it may be appropriate for more complex devices, such as those that are rechargeable. Balancing the tasks of delivering therapy, managing the battery and its charging, communicating over bluetooth, and possibly sensing biological stimuli is a lot. While bare metal programming can achieve all of this, I think that as therapy and battery management continues to increase in complexity, an RTOS becomes a better option. On

the other hand, in a device that is not rechargeable, all of the overhead of the RTOS' kernel may be more detrimental to battery life than its worth. I think this would again depend on the complexity of the therapy.