MTE 241 Lab 5 – RTOS Version: Prelude

*Created by Mike Cooper-Stachowsky, Summer 2022*

If your group chose to write the RTOS, you must complete this prelude assignment and prove that your RTOS can execute a context switch.

Contents

[Introduction 1](#_Toc106115038)

[Background 2](#_Toc106115039)

[What is a thread, really? 2](#_Toc106115040)

[Understanding the ARM Cortex M Series (a little bit) 3](#_Toc106115041)

[The important registers 3](#_Toc106115042)

[Handler Mode, Thread Mode, and the Two Stacks 4](#_Toc106115043)

[Working with the Stack Pointers 4](#_Toc106115044)

[OK, so what *is* a thread, really? 5](#_Toc106115045)

[That’s not very satisfying. How do we solve the bootstrapping problem? 5](#_Toc106115046)

[The Context Switch 6](#_Toc106115047)

[Exercise 1: There and Back Again 7](#_Toc106115048)

[Debugging an interrupt handler 7](#_Toc106115049)

[Exercise 2: The Context Switch 7](#_Toc106115050)

[What you need to submit 7](#_Toc106115051)

[The Appendix of Random Things 8](#_Toc106115052)

[Is Bootstrapping related to Booting a PC? 8](#_Toc106115053)

[What does the IP register actually do? 8](#_Toc106115054)

[Do stacks always grow down? 8](#_Toc106115055)

[Why is the order of the thread’s initial stack important? 8](#_Toc106115056)

[Why are we using PendSV and not a timer? 8](#_Toc106115057)

# Introduction

Creating your own RTOS is a fantastic learning experience. You will understand the very foundations of how an RTOS works, and you will be very well versed in the various problems that need to be solved when either writing or using a commercial RTOS. However, you should know that it is quite hard, the debugger will be your closest friend, and that you are highly unlikely to create a system that can compete with something like RTX. Approach this project with an open mind and a deep interest in learning and you will do well. Also, FYI, you are going to make the microcontroller hardfault. A **lot**.

This lab is not a cookie-cutter lab. You will need to do a lot of independent research. We are here to guide you, but not to do it for you. If you don’t know how a specific assembly mnemonic works, look it up. Read the documentation. Ask on forums (and Piazza, email, Teams…). This is a project that you get out of what you put in. You can definitely just find and copy a pre-made RTOS, but you will learn almost nothing!

You will need to set up your project on your own. For our purposes you will probably be able to get away with just setting up the project like you did in lab 1. However, any specific things you need to get your OS up and running are your personal choices. The only restriction is that you should not select an RTOS (as in, don’t enable CMSIS’s RTX5), since you are writing the RTOS itself!

# Background

In this prelude assignment you must demonstrate that you can execute a context switch. This is by no means a simple task, and we will guide you through a lot of the process. Once you can do a context switch the real design challenges begin. However, the exact implementation details are up to you, and some design choices you make may very strongly affect your ability to do certain things. Let’s look at what you need to know.

## What is a thread, really?

You know from class that a thread is an independently executing chunk of code. In labs we have typically used threads that follow the same basic format: a function that never returns and whose main body lives inside of an infinite loop. However, it can’t be that simple and it isn’t! Consider the following problem:

Let us say that we have a thread:

void thread1(void\* args)

{

while(1)

{

printf("Hello");

osYield();

}

}

This should seem odd to you already. If thread1 calls yield, how does another thread run? If that thread calls yield, how does the OS know that we are supposed to eventually return to thread1? If we have multiple threads, all calling yield, how do we keep track of which call to yield should be the one that returns us to thread1? How, in short, does any of this work?

Before we simplify we are going to complicate matters with yet another question – how do you start this in the first place? Let’s think about it for a minute – if thread1 is a function, then logically we should just be able to call that function and run the thread. But that simply won’t work! If it did work then we would never be able to call the function of another thread – thread1 never returns. You might think, then, that the yield function’s job would be to search for and start new threads. But if that’s the case, how would thread1 start in the first place? Does it contain the first call to yield? This is a very frustrating chicken-and-egg problem known more generally as “bootstrapping” (see the Appendix of Random Things to understand more here).

In general the problem of bootstrapping is solved by pre-loading memory with important information. For our operating system we will refer to this information as the thread’s “context”. To understand the idea of context we need to understand a lot more about how our microcontroller works.

## Understanding the ARM Cortex M Series (a little bit)

### The important registers

Note that this discussion is actually quite general – many computers you will interact with follow a similar design to the ARM Cortex M, with some being simpler and some being more complicated.

The ARM Cortex M, like just about any modern chip, is a “register machine” – the CPU does not access memory directly, and instead accesses only very specific memory addresses known as the registers. To begin building our multithreading system we therefore need to understand the registers.

There are 17 registers that the CPU can access and that are relevant to us (there are others, but we don’t care about them at this point). The first thirteen are labeled R0 through R12, and the remaining ones are very special with their own names. For our purposes, the following statements are true:

* R0 through R3 are known as “scratch” registers. You can overwrite them all you want, but that also means that you should never rely on their values being preserved across function calls
* R4 through R11 are “variable” registers. The compiler uses these for various purposes and in lab 5 you really, really shouldn’t mess with them.
* R12 is known as the “intraprocedural call scratch”, or IP, register. Its existence is 100% irrelevant to you except you need to know to never overwrite it. If you are interested in what it actually does, see the Appendix of Random Things at the end of this document.
* R13 is known as SP, the stack pointer. We are going to learn more about what it is and get really frustrated by it later.
* R14 is LR, the link register. For the most part it stores the return address whenever you do a function call. LR is going to be extremely important to our context switches, since it will tell us how to get back to a thread that had previously yielded.
* R15 is PC, the program counter. PC holds the address of the next instruction to run. PC is the key to starting a task without calling a function. Briefly – we store the value of the function we want to call in PC, and then voila! The function runs!
* In addition to these 16 registers there is another register, xPSR, which is actually an amalgam of several registers. “PSR” means “Program Status Register”. xPSR and its related registers are managed by the CPU and, although we don’t have to deal with it directly, we do need to know it exists since it gets saved to our stack when we do a context switch.

So why is all of this relevant? These registers will be used by the CPU to store information relevant to a thread. Therefore, when we want to switch between threads we need to store these registers for the current thread, then restore them for the thread we are switching to. How we do this is called a “context switch”, and the registers form part of the thread’s context.

### Handler Mode, Thread Mode, and the Two Stacks

The ARM Cortex M processor can run in two modes – Handler (used in interrupts) and Thread (used everywhere else). In addition to these two modes you can specify whether you are running in Privileged (access to the entire CPU’s instruction set) or Protected (access to a limited instruction set). In this lab we are not going to worry about Privileged vs. Protected – we’ll stay in Privileged mode forever. But we do need to know about Hander and Thread modes.

In addition to all of this, the Cortex M has a way to use two separate stack pointers. The Main Stack Pointer (MSP) is always used in Handler mode. In Thread mode you may choose between the MSP and a Process Stack Pointer (PSP). We are going to use the PSP to run our threads.

When running normal code that is not part of an interrupt we are in Thread mode. Thread mode is where all of your threads will run! When in an interrupt we are in Handler mode. Handler mode automatically and always changes your stack pointer to MSP and this has grave implications – we cannot use PUSH and POP instructions to save the task state. If we do, we are pushing and popping onto the wrong stack, and we will never be able to restore the context. This makes life irritating. Additionally, we need to perform some special steps to go back to thread mode when we are in handler mode. If we don’t the processor will hardfault. To make life harder, it is always advisable to do your context switching from an interrupt, so we have no choice but to use Handler mode for this.

In exercise 1 you will be triggering an interrupt and returning from Handler mode. Don’t skip it!

### Working with the Stack Pointers

MSP is set to a default value when your code starts running. If you want to know where it is, you do the following:

uint32\_t MSR = \*(uint32\_t\*)0; //MSR because MSP is a name we want to reserve

That’s right – you dereference 0! This isn’t as weird as it sounds: address 0 stores a lot of important processor information in a vector. The first entry in that vector is the address of MSP.

Stacks in the ARM Cortex M grow down (see the Appendix of Random Things for more information). Therefore, to create a new stack (for a thread’s stack that will eventually live in the PSP, for example) you can use the location of the MSP and subtract. For instance, you might do this:

uint32\_t\* thread1PSP = (uint32\_t\*)(MSR - THREAD\_STACKSIZE);

Note how the cast and brackets are handled and note that this relies on you to define the THREAD\_STACKSIZE constant. If your brackets are wrong the case will happen first, and then the subtraction will be in units of uint32\_t, not uint8\_t, which you may or may not want. That’s your choice.

Although it is possible to change where the MSP points to, don’t. You will, however, need to know how to change where the PSP points to. Since you only have one PSP but multiple threads, you need to switch the threads’ stack pointers out. To do this you need to do two things:

1. Set the control register to enable thread mode. This is accomplished via the following code:  
     
   \_\_set\_CONTROL(1<<1);  
     
   You only ever have to do this once as long as you only enter Handler mode via interrupts. As such, this is good to put at the first few lines of main and then never thought about again. It is why we just gave you the code.
2. Set the PSP via the following code:  
     
   \_\_set\_PSP((uint32\_t)pointerToMyNewPSP);  
     
   You’ll need to do this every time you want to change PSP, so at least once per context switch.

## OK, so what *is* a thread, really?

We are finally ready! We are finally ready to understand what threads are. Amazing.

At the bare minimum your threads require the following information:

1. A function pointer that is what runs when it’s the thread’s turn
2. A stack pointer that is used to change PSP and switch context

Although that’s technically all you need, you will likely find that you want more in your threads. For example you may want an ID, perhaps a memory location for thread-local storage (storage only your thread can access), priority, etc. The actual design of this part is up to you.

## That’s not very satisfying. How do we solve the bootstrapping problem?

Ah, this is the fun part – when we first create a thread we do not run it. Instead, we directly write to its stack to make it look like it has already been running! What this means is that you need some kind of createThread function. It needs to set the thread’s pointers and any other settings, then fill the thread’s stack in the following **very specific** order. Do not deviate from this order, seriously (see the Appendix of Random Things for why).

For the following, let’s presume that we have a pointer, sp, which points to the current location on the stack.

* First, we will be storing xPSR. This is usually set by the processor, but we need to ensure that its 24th bit is set. This specific bit represents which mode we are in, and it must be set if we are in thread mode. So the first line of thread setup code you write must be:  
    
  \*sp = 1<<24;
* Next, decrement sp. If your sp is a uint32\_t pointer decrement it by 1. Otherwise you need to figure out your alignment.
* Now store PC. If your thread function is called “threadFunction”, you do this by writing:  
    
  \*sp = (uint32\_t)threadFunction;
* Each time you add something to the stack you need to decrement the pointer. The next registers will be LR, R12, R3, R2, R1, R0, in that order. You do not need to set their values to anything specific but I recommend something obvious. For example, set LR to 0xE, R12 to 0xD and so on. This way if you use the debugger to figure out what is where in memory you can look for those specific patterns
* Finally, you need to set R11 through R4. Again, I recommend something obvious, perhaps counting down from 0xB or something.

Now…what? Let’s think about why this makes any sense.

Imagine that we went back to our original, naïve understanding of how to run threads – we would call the thread function to start a thread. If we did this, then the stack just before the function call executes would look exactly like this – xPSR would indicate we are in thread mode, PC would hold the function address we are about to jump to, and the rest of the registers would not be anything we cared about. So we are making the stack for each thread look like the function has been called, but it hasn’t yet! When we then execute the first context switch, it is the context switching function’s job to find a thread and load its context into the registers. Once we return from the context switch function we’ll be set and the thread will run.

## The Context Switch

Note: You are going to be stuck here for a long time. Context switching is the most sensitive problem to off-by-one errors that I have ever seen. However, in general a context switch is “simple”, in that describing what you have to do is easy but actually doing it is hard.

You begin a context switch with a call to some function, typically called yield but in Exercise 1 it is called trigger\_pendsv. This function’s purpose is to:

1. Determine whether a switch is even necessary – if there is only one task running, why bother?
2. If it is necessary, determine which thread to switch to
3. Save a useful offset of the current thread’s stack pointer somewhere that it can access it again once it is scheduled to run (see below)
4. Trigger the PendSV interrupt – we are using PendSV because it lets us control when the interrupt happens. Later on you will be using timers as well.

At this point xPSR, PC, LR, R12, and R3-R0 are pushed onto the stack for you by the hardware. Your stack pointer therefore is 8x4 bytes lower than it had been before you called the function. Keep that in mind.

1. Put R4 through R11 onto the stack
2. If you haven’t already, now is a really, really good time to save the thread’s stack pointer
3. Switch PSP so that it points to the new thread’s stack pointer
4. Pop R11 through R4 (notice the reverse order)
5. Return from the PendSV handler. The microcontroller will pop the rest of the registers out for you.

Here are a few hints:

1. When you are in the PendSV\_Handler, you cannot use PUSH and POP. These will push and pop onto the stack, which in Handler mode is MSP. We want things on PSP. Instead, you need to store PSP somewhere. R0 is a good bet. This looks like this:  
     
   MRS r0,PSP ;MRS loads PSP’s address into r0
2. The simplest way to load these registers is clumsy but it works: Use the STR instruction to store the register into the address pointed to by R0, then decrement the address stored in R0 by the correct number of bytes. You may consider looking into faster functions, like STMDB (“Store Multiple, Decrement Before”, which automatically decrements your address for you and stores all of the registers at once), but they require more reading to understand how they work
3. You will probably want to write a C function for handling the actual switch between the various thread stack pointers, but this can be done in assembly as well.

One of the hardest parts of a context switch is figuring out what to store as the stack pointer in the thread’s data. This is because we aren’t pushing and popping, but are instead placing things on the stack indirectly. If we could use push and pop we’d be trivially done – just push things and your SP gets updated, then pop them and your new SP gets updated. Exactly what you do and how you do it is up to you. Personally, I stored the eventual address of the thread’s SP in step 3. This meant that my thread’s data structure stored the current SP minus 17\*4 – I was storing 16 registers, each 4 bytes long, but I was using STMDB, so that “decrement before” thing meant I had to go one further. Then I called my switching function and the actual stack was decremented so that it aligned with what I stored. You do you. The point is that you need to keep track of things. If you are wrong your LR will not be properly restored, which means that your thread will not return to where it was when it called yield, and you will hardfault.

# Exercise 1: There and Back Again

This is technically optional but if you skip it and then get confused why things aren’t working I will scoff at you.

We mentioned in the background section that we need to do our context switch inside of the PendSV interrupt. In this exercise you will trigger that interrupt and correctly return from it. This must be written in assembly, even though technically this exercise will work in C.

1. Add “exercise1.s” to your project
2. On line 7 we tell you to load a constant into LR. Do that now. You will need to learn how to load immediate values into registers to do this. This is a special constant that, when loaded as a return value, tells the processor to return from Handler mode, go back to Thread mode, and use PSP as the stack pointer.
3. Use the function trigger\_pendsv to trigger this interrupt. If the processor doesn’t hardfault, congratulations!

### Debugging an interrupt handler

Like we said above, the debugger is your closest friend. However, you cannot just step line by line and hope that the interrupts happen. They won’t. This will cause a problem where your actual code, if it were running without the debugger, would correctly execute the interrupt but the debugger prevents this and your code faults only while debugging. To see if interrupts are working you need to put a breakpoint at the first line of the interrupt handler then allow the debugger to run the code. Once the interrupt triggers the breakpoint will as well and you can step through your code again.

# Exercise 2: The Context Switch

Create a minimal system that is capable of switching between two threads. What exactly these threads do is up to you.

You may consider setting the PendSV interrupt’s priority so that it has the highest priority of all interrupts. To do this you use a bit of magic:

/\* set the PendSV interrupt priority to the lowest level 0xFF,

which makes it very negative\*/

\*(uint32\_t volatile \*)0xE000ED20 |= (0xFFU << 16);

In this prelude you don’t need to do this, but if you eventually want to run multiple interrupts this is important.

# What you need to submit

Submit a brief (no more than 2 pages) report to Crowdmark that includes code snippets and discussions to prove that you have successfully achieved context switching. PuTTY screenshots are good but they are not enough – we need proof that what we are seeing in the screenshot is actually a context switch and not just two printf statements being run in series.

If we are satisfied that you can achieve a context switch we will allow you to proceed with the RTOS lab. Otherwise you will need to do the game.

# The Appendix of Random Things

## Is Bootstrapping threads related to Booting a PC?

Similar but not quite the same. A PC with an operating system has a slightly different but related problem – in order to load programs you need a program called a loader. It is responsible for finding the programs on the hard drive, loading it into memory, and jumping to the start. But the loader is a program! What is it that loads the loader? We need some way to bootstrap program loading in much the same way we need to bootstrap thread loading. The term comes from the idea of “flying by pulling yourself up by your bootstraps”. It is a circular problem that is usually solved by hard coding something. In a PC you typically hard code a region of memory to be “the” start address for your chip, then burn code into some kind of permanent storage. The PC starts by looking at that specific address and just runs code that lives there.

In Linux, a special file called initramfs (formerly initrd) is loaded into RAM early in the boot process. initramfs is a pre-made file system that contains a bunch of programs that are already ready to go. It is loaded by the bootloader or some descendent of it, which itself ultimately lives at that hard-coded memory address we talked about in the previous paragraph. Windows does something similar, and osX should never be used.

## What does the IP register actually do?

When we call a function we need to save the return address somewhere. This is typically done in the LR register. However, using LR to return from a function brings problems because the instruction used to do that can only access a subset of the bits of LR. Therefore, another register is sometimes needed to provide access to the full memory space. IP is used by the compiler to do this. Generally speaking, don’t touch IP and you’ll be OK.

## Do stacks always grow down?

No. It is more correct to say that the Advanced Arm Procedure Call Standard (AAPCS, available here: <https://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/ARM-AAPCS-EABI-v2.08.pdf>) enforces that the stack starts at a high address and grows to a low address. This is not a requirement of the processor because it is not necessary to use a stack. However, unless you are doing something very, very weird, your code should conform to the AAPCS and therefore your stacks will always grow down.

## Why is the order of the thread’s initial stack important?

This comes from the AAPCS again. When a function call occurs, the registers xPSR, PC, LR, R12, R3-R0 are pushed onto the stack in that order by the hardware. If you deviated from the order then you might try to load xPSR with the PC or something else. Since we don’t have any power over this order, we must respect it in our code. It is not a design choice, it’s a requirement.

## Why are we using PendSV and not a timer?

You’re going to later! We are using PendSV because it lets us control when the interrupts happen.